UNIVERSITY OF OSLO
Department of Informatics

Adaptive facial behaviour using selected methods in machine learning and motor control

Master thesis

Aleksander Paus

01. Mars 2006

Abstract

In the not so distant future, androids may be part of our everyday lifestyle. There can be a desire not only to make these android do our work, but also to enable communication with humans in a natural way. As a substantial part of human communication is through our body language, natural mimic is essential if artificial communication is to seem real. Even though this might appear like a trivial problem, there are a lot of obstacles to solve.

To achieve physical human appearance, we have to develop artificial skin that looks and folds naturally. This is far from an easy task, as living tissue has totally different characteristics from synthetic materials. Secondly we need some form of artificial actuators, preferably situated in or behind the synthetic skin. Human muscles have some amazing properties that we are not able to match yet. They are silent, strong, flexible, and precise, and last for millions of cycles. At last there is a need for a sensory system of some sort. Humans has an extremely advanced feedback system, providing information about factor such as pressure, temperature, pain etc. This feedback enables humans to make advanced decisions about their surroundings, and adjust their appearance accordingly. Even though all these factors were in place, natural mimic wouldn't be achieved before the android developed adaptive behaviour of its face expressions. A robot could of course be pre-programmed with a fixed set of face expressions, but this would undoubtedly restrict the personification of such an android. In order to achieve natural mimic and the impression of personality, its essential to make the androids facial expressions adaptive and to make the android learn and create face expressions never shown before.

This master thesis addresses some parts of the problem of achieving adaptive facial behaviour, essential for making androids operate in social settings, and obtaining natural communication with humans.

Contents

Chapter 1

Introduction

The target with this master thesis is to give insight to the subject of letting a robot achieve as natural communication as possible with humans. The subject regards the facial expressions generated by an android, and how these can be adjusted or adapted in order for the android to communicate in a natural way.

A future scenario

Suppose you were standing in front of full sized human robot with an artificial face, and it raises its hand to greet you. It smiles and introduces its self with a welcoming nice voice. Suppose now that you were unhappy, and have a hard time trying to put a smile on your face. The robot stops smiling, and looks back at you with a compassionate expression and asks with a polite voice if there is something wrong. It would certainly be difficult to feel that the machine in front of you didn't care about your feelings, even though the robot has no feelings what so ever. The android only needs to analyze your face, reckon that this person has a sad expression, adjust its own face and voice, and ask a preprogrammed sentence for sad face expressions.

When or if this scenario ever will become reality is hard to tell, but there are nevertheless numerous task in the society that could be solved by having human like androids. The more human like behavior they can achieve, the more tasks they can be set to perform.

Human communication

To achieve the goal of emulating human communication, there is a tremendous amount of research that has to be done distributed amongst several different technological and psychological branches. This master thesis deals with just a small fraction of some of these problems, namely how to create an android with adaptive facial behaviour. To understand why this is so important, it is necessary to realize that a great deal of our communication is through our body language. Professor Albert Mehrabian's concluded in his book "Silent messages" that 55% of our communication is non verbal[23]. Surely not all of this is done through facial expressions, but one can safely say that facial expressions constitute an important part of our non verbal communication.

1.1 Reasons for having adaptive face expressions

If the future enables us to create androids with true human looks and appearance, adaptive facial behaviour will become an essential part to achieve natural communication. It is important to realize that if androids are to look real, they have to be made up of materials with properties close to human skin and tissue. These material would most probably have dynamical properties as the human body is a dynamical system. A android with true human looks using only pre defined face expression would

therefore seem unnatural and stiff for several reasons. Roughly spoken, there are four main reasons for an android not to only have pre defined facial expressions.

1. Fatigue changes the characteristics of the muscles and the skin. The outcome of a pre defined face expression can therefore be changed if a static muscle activation is used.

2. Temporary noise such as temperature, or temporary mechanical failure might disrupt a pre defined face expression. By temporary means that when the disruptive factor is gone, the pre defined face expression would look the same way as before.

3. Tear and wear on the materials will change the characteristics of the face, disrupting a pre defined face expression.

4. The android might wish to generate a new face expression in order to imitate or adapt to its surroundings.

While these tasks can be quite difficult to solve for an android, humans copes with them automatically.

1.2 How humans achieve adaptive facial behaviour

Roughly spoken there are two main mechanisms that enable humans to have adaptive facial behaviour. First of all, it is important to realize that human being has got a highly complex nervous system, continuously providing fairly accurate feedback about a wide range of attributes, such as pressure, temperature, pain, etc. This feedback enables the brain to create a model of how we work. We walk around with a built in model of almost everything we do without even thinking about it. As an example, a human being knows how to touch the nose tip with its eyes closed. These models are in neural science and motor control called internal models [9, 6, 24], and are an important part of this thesis. The human body is a dynamic organism, and with aging our muscles and tissue changes its characteristics, and so does the environment for the internal models. To make sure that we still can perform the same maneuvers, the internal models are continuously being updated. The ability for an android to create such adaptive internal models is therefore essential if they are to achieve natural mimic under varying conditions.

1.3 What technology is needed

There are several technological issues that needs to be addressed for androids to mimic adaptive facial behaviour. Nevertheless, five main factors are especially essential:

1. Human like skin and tissue

2. Human like actuators

3. Precise feedback system

4. Internal models

5. Processing power

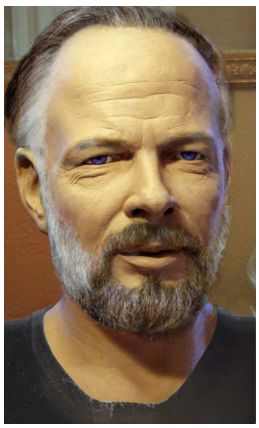


Figure 1.1: The Philip K. Dick android

1.3.1 Artificial skin

Artificial skin in the form of rubber latex and silicone rubber has existed for a long time in the film industry [29]. These material are not specially made for creating androids with human facial movement. However, David Hanson, the CEO and man behind Hanson Robotics has recently developed a new type of artificial skin called Frubber^{TM}. David Hanson is considered one of the best android head builders in the world, and won the 2005 AAAI[1] competition for creating the best android of Philip K. Dick [13]. The android was also one of the main installations at Wired's Nextfest in June 2005. Figure 1.1 shows the android that will be a powerful memorial to the famous science fiction writer. Even though Frubber^{TM} is a great improvement from materials such as rubber latex [13], the characteristics of these materials are still far from human tissue. Hopefully the future will give us new materials that resemble our skin to a greater extent. Whether these materials will be synthetic or some sort of living tissue, they will most likely degrade over time, and mechanisms for compensating for this tear and wear will be necessary, in order to make facial expression look realistic.

1.3.2 Artificial muscles

The human body use muscle fibers as actuators. Human muscles have got a merger of properties that modern technology hasn't matched yet. They come in almost any shape and power. They have a high yield ratio, while being silent and precise. They last for millions of cycles, and are both deformable and self repairing to some extent. While modern technology has given us actuators that are far better in some areas, the combination of these properties are only possessed by biology. Engineers are therefore constantly on the search for materials and techniques that can imitate the biological characteristics of muscle fibers. Artificial muscle fibers are a large group of actuators that all have the property of retracting under certain circumstances. They exist in many shapes [44, 11, 21], from electro active polymers, to wired metal that contract when voltage is applied. So far, no one has come close to the real thing, but nonetheless we have developed interesting technologies that exhibit some of the characteristics possessed by human muscles. The future will hopefully give us new materials that can mimic more of these features.

1.3.3 Feedback system

As we shall see, the type of feedback available affects both the possibilities and performance of creating natural mimic. Looking back at how humans get their feedback,

[1] American Association for Artificial Intelligence

we can identify three main sources.

1. The nervous system
2. Feedback from other people
3. Mirrors or cameras

To a large extent, these feedback mechanisms can also be interpreted by androids. Feedback from other people or androids might be of relatively low relevance, as the accuracy of this feedback is low. However, it can signal whether the face expressions performed by the android are within an acceptable range or not. From a technological point of view the most important part is therefore what kind of feedback can be obtained from sensors in the skin or through a visual system.

When can corrections be done?

Given a precise sensory system in the skin, an android could in theory correct its face at all times. The question will be whether these corrections can be made in social settings without being noticed by its surroundings. As we shall see, this depends to a large extent on the properties on its internal model. A visual feedback system on the other hand can only give feedback when it is in use. Corrections to the androids face can therefore only be made when the android is in front of a camera or mirror, and thus sets restrictions to how often an android can perform updating.

It is therefore important to make a clear distinction between visual feedback and feedback from sensors in the skin. Visual feedback is sporadic, while feedback from sensors in the skin is continuous.

Preciseness

Another important question is how precise the feedback system needs to be. Looking at artists drawing caricature, it can seem that only certain structures such as wrinkles and eye brows are necessary to capture the essence of a specific person, or a facial expression. However, communication between close relatives can be of a different character, and the ability to see whether your loved ones is lying or not, can be based on much smaller facial movements than just the position of the eye brown or wrinkles in the forehead. A precise feedback system could therefore be of great value if the android is to look very realistic.

Type of feedback

While it is easy to obtain the position of different structures in the face using visual feedback, the same task becomes substantially more difficult using sensors in the skin. If feedback from sensors in the skin are to give any information about the position of different face structures, there has got to be a mapping between the each sensor's position, and the resulting skin position. However, if such mapping is possible to obtain, sensory feedback could use three dimensions to describe the position of the skin, whereas this becomes more difficult for visual feedback.

In the simulator created for this thesis, introduced in chapter 2, we just assume that exact feedback is possible to achieve, and will be available in two dimensions. However, we make a clear distinction between visual feedback and feedback from sensors in the skin, in the way that sensory feedback is regarded as continuous, while visual feedback is sporadic.

1.3.4 Internal models

In neural science and robot control, the notion of an internal model has emerged as an important theoretical concept [7]. An internal model is capable of predicting the

response of performing an action without executing the action. These predictions can therefore be used to control or guide a system in order to perform specific tasks. If a human picks up an apple to grab a bite, the brain has to predict how to pull each muscle to perform this action. The phrase internal model is used to distinguish the actual transformation performed by the muscles, and the virtual representation of the same task in the nervous system.

The ability and difficulties with creating internal models will be a main focus in this master thesis. Before dwelling into the details of this, we can just say that the ability to create accurate and adaptive internal models is imperative for creating adaptive facial behaviour.

1.3.5 Processing power

The human brain is a massive parallel processing system. Many of the task performed by humans involves huge computational activities. As this subject of this thesis is of future relevance, we can assume that processing power has increased substantially, and that tasks that are time consuming today, will be more or less negligible.

1.4 Different levels of adaptive facial behaviour

Depending on what technology the future brings, we can achieve different levels of adaptation. The main difference will be whether or not one demands that the android is able to correct/adjust its face in a social setting or not. The technological demands for an android that can adjust and change its face expressions without visible trial and error are much more stringent. One can divide the level of adaptation in four categories.

Non adaptive

The simplest level of realism would be an android with only pre defined facial expression. If no mechanisms are incorporated to achieve adaptive facial behaviour, the android would have no possibility to correct or adjust its face expressions due to malfunction, tear and wear or other factors. Generating new facial expressions by imitating other people will also be impossible. This is to large extent were we are in the year 2006.

Occasionally adaptive

As we have mentioned, an android relying on visual feedback can only make corrections to its face when this feedback is given. As long as noise or malfunction is held to a minimum between updates, an android relying only on visual feedback could in theory both create new facial expressions and adapt to tear and wear. However, solving the issues of temporary noise and fatigue would be difficult, as these problems most probably would appear when visual feedback is absent. An android relying on visual feedback could therefore only be occasionally adaptive.

Continuously adaptive, but not in social settings

If the android instead of visual feedback relied on a precise sensory system in the skin, corrections could in theory be performed at all times. However, it is important to realize that without a perfect internal model, the android could make mistakes during updates. Corrections to the android face should therefore not be done in social settings, as unnatural grimaces would disrupt the feeling of natural communication. If malfunction and noise is not flourishing, an android could in theory cope with all the four reasons for having adaptive facial behavior mentioned in section 1.1. However,

the value of a continuous feedback system will depend on how much time is spent on updating the face.

Continuously adaptive under all scenarios

In order for the android to be adaptive under all scenarios, one cannot allow the robot to do any mistakes, unless these are so small that they will go unnoticed. To achieve this goal, there are two main factors that has to be present.

1. Continuous and accurate feedback from the skin

2. An advanced internal model that is able to immediately make correct predictions as to how to correct the face.

The question becomes how to instantly cope with problems such as temporary noise, or malfunction due to tear and wear. If noise is defined as factors not included as parameters in the model, the internal model must have some mechanisms to immediately adjust itself in presence of unknown parameters. If such a model could be created, it could of course cope with all the reasons for having adaptive facial behaviour mentioned in section 1.1.

1.5 Outcome

Up until now, no one has focused on achieving adaptive facial behaviour. Researchers are still paying attention to the practical issues of creating android heads with realistic skin and natural facial movement. However, with better materials and advancement in robotics, adaptive facial behaviour might become a demand in the future. We have put adaptive facial behaviour on the agenda for the first time, and looked closer at the question from a computational perspective. Originally, the idea was to create an android head in hardware with artificial muscle fibers. Unfortunately this implementation raised a number unforeseen problems, so a skin simulator was created instead. The skin simulator is introduced in chapter 2. Our simulator has enabled us to show that:

1. Optimization algorithms can be used to create androids with adaptive facial behaviour, but not in social settings

2. If continuous feedback can be given from sensors in the skin, internal models could be used to increase the performance of optimization algorithms

Optimization algorithms

Under the premise that exact feedback can be given, we have looked at optimization algorithms as a way to achieve adaptive facial behaviour. However, an android relying on algorithms has to perform adjustments online, i.e. through trial and error. We have looked at several promising candidates to see what mechanisms could be used to solve the task. We have come up with a new algorithm named Push or Pull. The algorithm has its weaknesses, but it has turned out to be quite efficient compared to more familiar methods such as Genetic Algorithms.

However, we have also realized the opportunity in using video stream as feedback, and opened the challenge for others to use continuous visual feedback as a way to speed up the performance. The implementation and results of using the algorithms are explained in chapter 3.

Internal models

If continuous and precise feedback is available through a sensory system, we have shown that internal models could be used to reduce the number of online trials. Chapter 4 describes the implementation of our internal model and the results obtained. We have also found some limitations in our implementation, and research in the field of motor control and internal models has given us some clues as to how the internal model can be improved.

A new scheme

Based on the research in the field of motor control we have come up with a new scheme built on the MOSAIC [7, 5, 45, 1] architecture. This new scheme could in theory be used to create android faces that are continuously adaptable under all scenarios. Our new scheme and improvements to our model is introduced in chapter 5.

Chapter 2

The workbench

2.1 The android head

Originally the objective was to build an android face in hardware. This android head was to be used as a workbench for studying adaptive facial behaviour. A rubber latex mask was to be used as artificial skin, while a camera was to be set up to produce feedback to the face in absence of an adequate sensory system. To make the face more realistic, we wanted to use artificial muscles instead of servos as actuators. Even though servos are the common way of creating actuators in android heads today [3], they create a lot of noise, and absorbs a lot of space. Figure 2.1 shows the inside of one of David Hanson's robots. This android face has only got 24 servos[13], and the more realistic face that is to be created, the more servos are needed. To cope with this problem, we wanted to use Flexinol® wires as actuators.

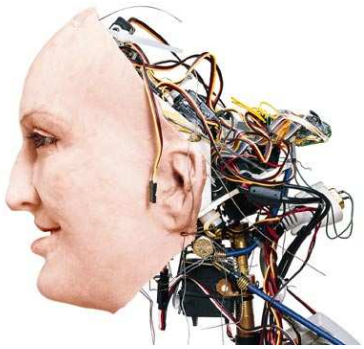


Figure 2.1: Android head with 23 servos created by David Hanson.

Flexinol®

Flexinol® [11] belongs to a group of metals called shape memory alloys (SMAs) [22, 33]. Shape memory alloys differ from other metals in many ways, but most importantly because they have the ability to undergo a phase change while being in solid state. The phase change is triggered by temperature, and makes the metal useful in a wide variety of applications. Flexinol® is a shape memory alloy specialized for the use as actuators. By sending current through a Flexinol® wire, the wire is heated and forced into a state transition. As a result, the wire contracts. Contraction is relatively spontaneous, and is only limited by the time it takes to heat the wire. Normally this can be achieved in less than a second. Flexinol® wires are light and silent, and used the right way, they can obtain repeatable motion for tens of millions of cycles.

Obstacles

Unfortunately the approach using Flexinol® turned out to have some severe flaws for a number of reasons. The actuators needed to be close to one meter long to achieve desired movement and speed. They therefore had to be placed along the spine, both due to these physical restrictions, but also due to their excessive thermal development. As Flexinol® contracts due to heating, their thermal development would affect each other in a way that would be hard to predict. The entire machine needed to be very rigid, and placed under constant lightening conditions to simplify the image analysis, and avoid too much error from the camera. Tear and wear on the tendons (nylon wires), connections and skin also raised a number of problems.

To cope with these challenges, we realized that the head had to be simplified in a great deal before we could be able to study adaptive facial behaviour. A lot of effort would have been spent on reducing the amount of non relevant noise, i.e. noise originating from poor implementation of the android head. Besides, a lot of the human-like appearance of the hardware model was removed as the muscle fibres had to be placed in the spine.

After trying to create a prototype of the android face in hardware, the decision fell on creating a model of the skin in software. The creation of the simulator in software reduced development time, and enabled more focus on the theory of the subject instead.

2.2 The simulator

The simulator was written in C#, and is a windows application that enables the creation of arbitrary facial structures by drawing muscles and points to a skin surface. The skin is visualized by an array of dots with a brown colour. The dots can be regarded as the atoms comprising the skin, even though they are not atoms in the normal sense. The program let the user select the density of the skin by setting the number of dots in each direction. The resolution of the skin can also be adjusted, depending on the needs for precision in the simulator. If a face is created with a resolution of $500 * 500$ pixels, then each atom in the skin can have up to 500 different values in the X and the Y axis direction. Figure 2.2 shows the user interface of the simulator along with properties window, showing some of the settings available.

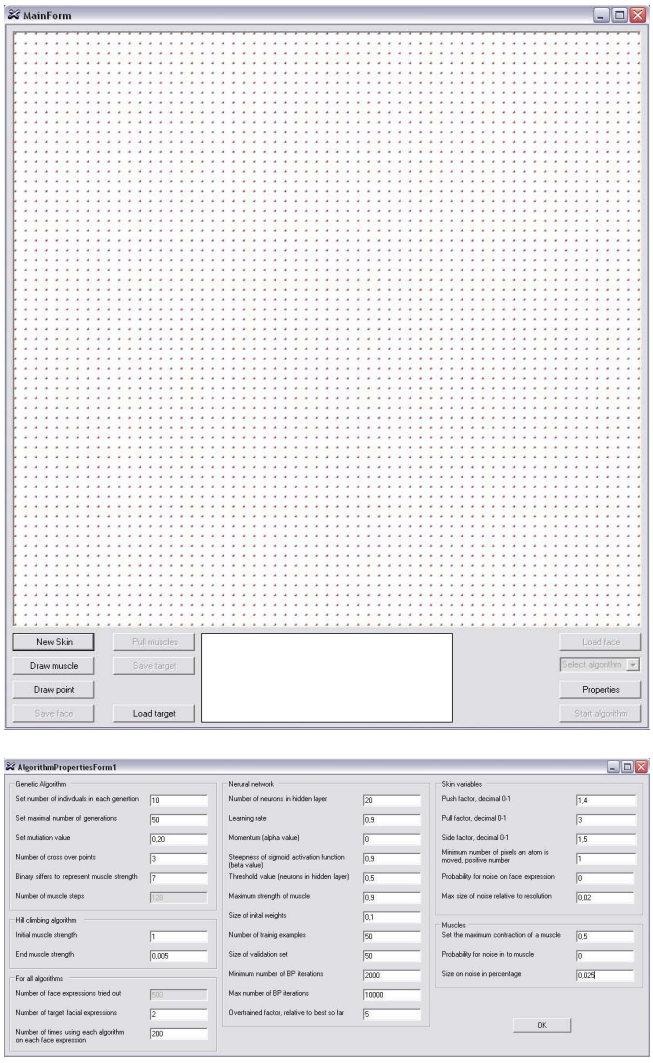


Figure 2.2: The user interface with the properties window below

By using the mouse cursor, the user can draw muscles and structures to the skin. The structures are coloured blue while the muscles are coloured red. Each muscle is connected to the skin at one end, and to the skeleton at the other end. Figure 2.3 shows a screenshot from the simulator, imitating a simplified eye brown with two muscles acting on it.

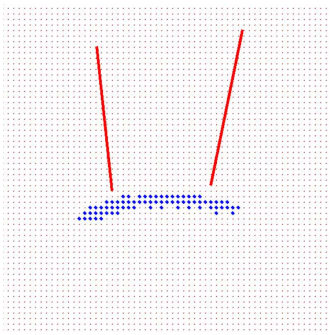


Figure 2.3: A simplified eye brown with two muscles acting on it

The muscles

Each muscle take as input a strength value between 0-1, where 1 is maximum strength. A nonlinear function is thereafter used to calculate the force excreted by the muscle. As we are not able to predict the physical properties of future materials, we have based the muscle function on results obtained from human muscles. Creating a model for human muscular activation is a highly complicated process. Different models have been proposed [12], but they all include many factors and variables that would be too complex to implement for this simple simulator. However, under certain circumstances, the force exerted by a human muscle can be modelled as fairly linear relationship to its neural activation [15]. We have therefore chosen to implement a muscle function with a close to linear sigmoid shape. Figure 2.4 shows the muscle function. Even though this might be a great simplification of how future materials behave, internal mechanisms inside or outside the muscle could be used to create a close to linear relationship between the desired force created, and the signal used to activate this force.

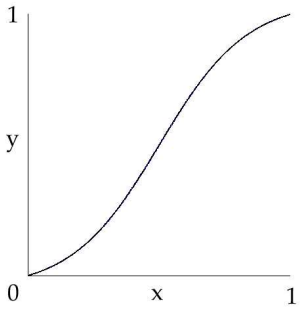


Figure 2.4: The muscle activation function. Muscle activation signal is along the x axis, while the resulting force is along the y axis

The skin

The muscles are the only forces acting on the skin. When a muscle is contracted, the individual atoms in the skin are updated according to their angle and length form the muscle. If there are several muscles in the simulator, the position of each atom will be a result of the sum of forces applied by all muscles. The atoms are affected differently depending on whether they are in front or behind the muscles when they are contracted. The functions updating the atoms in the skin are all exponential, but the degree of non linearity is adjusted so that the skin would looks as realistic as possible.

Target positions

After drawing the muscles and the structure in the simulator, each muscle can be pulled in order to move this structure to a new position. This new position can then be stored by the program to yield a target position. These target positions can be looked upon as the target face expressions the android want to reach. As mentioned in section 1.1, this could be either because:

1. Fatigue changes the characteristics of the muscles and the skin
2. Temporary noise disrupt a pre defined face expression
3. Tear and wear on the materials have disrupted a pre defined face expression
4. The android wish to generate a new face expression in order to imitate or adapt to its surroundings

Figure 2.5 shows a target position for the eye brown in figure 2.3. The blue points have turned green to differentiate the target expressions from the structure's initial position. The initial position is the location of the structure when all muscles are relaxed.

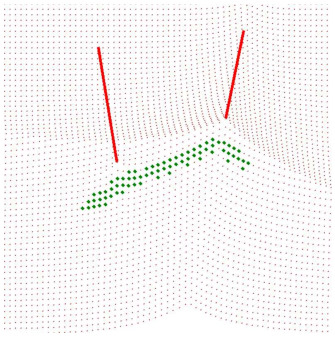


Figure 2.5: Target position for the eye brown in figure 2.3

Error measure

By definition the initial position has an error of one, while the target position has an error of zero. The goal is to pull the muscles so that all the points in a structure are aligned with the position of the target expression, thereby yielding zero error. The error of each specific face expression, i.e. a certain muscle configuration, is calculated by summing the distance of each point in a structure to the target expression. If the distance from the initial position to the target expression is 100, then a muscle configuration that results in a distance of 50 will have an error of 0.5. It is important to emphasize that each point's distance is measured in absolute values. If the muscles are pulled too hard, and the structure move past the target position, the error will still be positive. This simple error measure is used to avoid complicated image processing. Other means than using a collection of points as feedback are of course possible. However, all structures can be created by a collection of points, justifying our simple implementation. The error obtained when contracting the muscles can therefore be looked upon as feedback from the feedback system introduced in section 1.3.3.

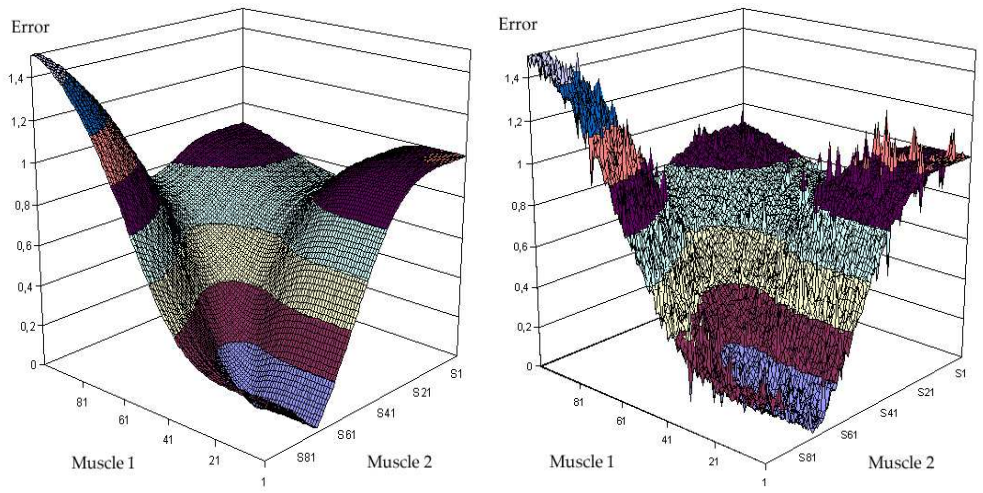
Error space

For each target expression, every muscle configuration has its own error value. By pulling each muscle by a small amount at a time, one could read out the error from each

muscle configuration. The error obtained could be used to create an (N+1) dimensional error space for the face, where N is the number of muscles. Figure 2.6(a) shows the error space for the target position in figure 2.5. From the seemingly simple task of moving one structure with two muscles from the initial position to a target state, we see that the error landscape is quite complicated. The error landscape has one global minimum and several plateaus and valleys. Even though this error landscape is only possible to visualize for two muscles, its valid for any number of muscles, creating an increasingly more complicated error space.

Noise

To make the physics in the simulator more realistic, noise is introduced from two sources. The most frequent source of noise is from the muscles. When a muscle is pulled, the signal to the muscle is corrupted, yielding either lower or higher strength than the intended signal. The second source of noise comes from the feedback. If feedback noise is introduced, all points in the face are moved in the same direction by the same amount. Figure 2.6(b) shows how the noise added in this simulator affects the error landscape of figure 2.6(a).



(a) Without noise

(b) With noise

Figure 2.6: Error space of the target in figure 2.5. The error space is created by pulling each muscle from zero to max strength, and reading out the error for all resulting muscle configurations. The muscles strengths can be read in %

Chapter 3

Creating an adaptive face without an internal model

Even thought internal models might be a smart way to go in order for an android to achieve adaptive facial behaviour, they might be quite complex and difficult to engineer. An easier way would be to abandon the idea of creating such a model, and just rely on other means to correct imperfections or to generate new face expressions.

3.1 Using algorithms to achieve adaptive facial behaviour

Assuming that the android can obtain correct feedback of some sort, one easy way of correcting errors could be to update the face from time to time using algorithms. The problem is that without an internal model, the android has no clue about the mechanics of neither skin nor muscles, and can therefore only correct or generate new face expression through trial and error. This of course could look rather obscure, so updates should be done when the android is alone. Therefore, only relying on algorithms cannot make the android become adaptive under all scenarios.

What can be solved

Looking at the reasons for not having only pre defined face expression introduced in section 1.1, we realize that an android relying on visual feedback and algorithms could in theory cope with problem 3 and 4. However, adjusting for temporary noise and fatigue would become difficult when updates only are performed occasionally.

If continuous feedback is given from sensors in the skin, all problems except fatigue could in theory be corrected. However, the usefulness of correcting for temporary noise would depend on the time needed to perform corrections, and the timespan of the temporary noise.

To see the effect of using algorithms to achieve adaptive facial behaviour, we have implemented and tested a handful of algorithms on our simulator.

3.2 The challenge created by the simulator

In our simulator, the task for an algorithm is to reduce the error obtained from the initial position when all muscles are relaxed. As the target expression has got an error of zero, the problem is to pull the muscles so as to align the structure with the target position. Looking at the error landscape in figure 2.6, we realize that the problem is to find the global minimum of the error space. The task can therefore be classified as optimization problem [43]. However, it is important to realize that feedback to the

algorithms can only be provided after the muscles have been pulled and the resulting error calculated. In a real world scenario, pulling the muscles to yield a new face expression would take a considerable amount of time. The goal is therefore not only to find the global minimum of the error landscape, but also to do this in the least number of trials. It is important to recognize that all target positions are created in a way that is possible to reach. In other words, at certain muscle strength, the structure will reach its target position, so all problems are solvable.

The error space

Looking back at figure 2.6(a), we see that there are several flat regions and valleys, but no local minima in the error landscape. By taking a cross section of figure 2.6(a) we can read out the error function for one muscle, shown in figure 3.1. As we see, this error function has only got one minimum. Remember that the error is calculated by summing the distance of each point in a structure to the target expression. Pulling or relaxing the muscle from the strength yielding the lowest error will only result in the points having the same or increased distance. The error function for each muscle will therefore only have one minimum, and the error space for the facial structure will consequently have no local minima. However, the introduction of noise adds temporary local minima to the the error landscape as can been seen in figure 2.6(b).

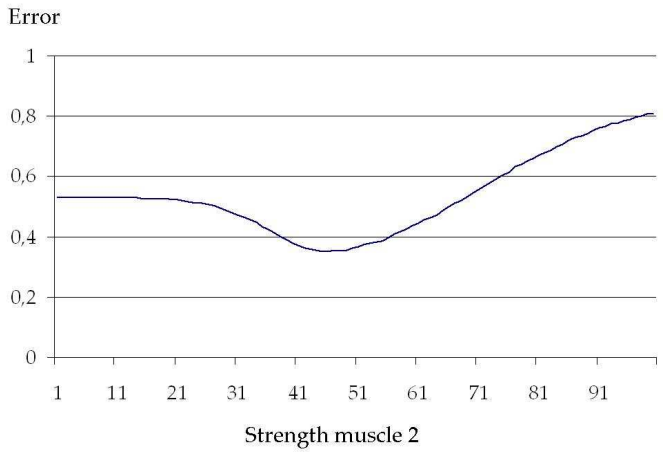


Figure 3.1: Cross section of figure 2.6(a) when muscle 1 is held at 55% strength

3.3 The algorithms

There are many different algorithms that could be used to descent the error spaces created in our simulator. Wikipedia lists over 40 different algorithms under the category "Optimization algorithms" [43]. However, most of these algorithms are not applicable for our problem. Interesting candidates in this list includes:

- Differential Evolution [10], Genetic Algorithms [36], Golden Section Search [37], Hill Climbing [39], Simulated Annealing [40], Tabu Search[30].

Inspired by some of these methods, we have implemented five different algorithms to see the effect of using optimization algorithms to create adaptive facial behaviour. The algorithms all have different ways of descending the error space. To compare their performance and efficiency, they are therefore exposed to several problems with and without noise. Each algorithm is run on three different face structures, i.e. different constellations of muscles and points, and has to reach four different target facial expressions, totalling to 12 different problems. To yield comparable results, each algorithms starts off with a random pull, and has to descend the error landscape in the least number of tries to reach the global minimum. The theory and implementation of the different algorithms are introduced in the following subsections, while the results obtained by running them on the simulator are shown in section 3.6.

3.3.1 Stochastic Search

Stochastic Search is an explorative algorithm in its simplest form. Stochastic in this setting means random, and is basically an algorithm that tries to find the minimum error by iteratively suggesting a solution to the problem by random.

Algorithm 1 Pseudocode Stochastic Search

```
for (the number of trials)
  give muscles random strength and calculate error
  if (error < best error so far)
    store muscle strengths
endfor
```

Stochastic Search iteratively pull the muscles by random, but stores muscle configurations that yields a lower error.

Stochastic Search is complete, i.e. it will sooner or later find the best solution a problem, given a hypothesis space of a finite size. However, the efficiency in the long run can not be expected to outperform simple enumerative search. That is, going through all possible values in the fitness landscape one at a time. Nevertheless, Stochastic Search is totally unaffected by noise as it makes no use of feedback at all. Its simplicity and robustness therefore makes it ideal for benchmarking other algorithms.

3.3.2 Hill Search

Hill climbing algorithms [39, 26] are a class of algorithms that analyse the error space in close proximity, and move in the direction of steepest gradient. Hill Climbing can find the steepest gradient by differentiating the error landscape, or by testing out all possible solutions in the immediate proximity before deciding where to move. In our problem, the function of the error landscape is not known, leaving us with testing out nearby solutions to find the gradient. However, there are two reasons for not implementing Hill Climbing the original way:

1. Standard Hill Climbing makes use of the error values in close proximity, and in our problem, these values can be affected by noise, giving meaningless feedback to the algorithm.

2. Calculating the error of all surrounding solutions requires a lot of trial and error, growing exponentially as the dimensions of the error space increased. In an error space for two muscles, trying out all possible solutions in close proximity would require $(3^2 - 1) = 8$ tries. I there instead where 6 muscles involved, each update would use $(3^6 - 1) = 728$ trials.

In order to cope with both these problems, we have implemented a different strategy for the algorithm which we call Hill Search. Instead of trying out all possible values, it tries out all dimensions before making a move. Besides, it starts off with testing distant values, instead for trying out solutions in close proximity, and reduces its search area over time.

Algorithm 2 Pseudocode Hill Search

```
give muscles random strength and calculate error
for (each iteration)
  STEP = 1 / iteration
  for (each muscle):
    temporary add STEP to the current muscle strength and calculate error
    temporary subtract STEP from the current muscle strength and calculate error
  endfor
  if (error from contracting or subtracting STEP to any muscle < best error so far)
    add or subtract STEP to that muscle
    store muscle strength
endfor
```

The algorithm starts of with a random pull. From here, it goes through all muscles and adds and subtracts STEP to each muscle. If any of these muscle configurations yields a lower error, the algorithms stores the strength of the muscle that gave the lowest error. The STEP value is reduced for each iteration, i.e. every time all muscles has been pulled. The reduction of the STEP value is hyperbolic, i.e. $1/X$ where X is the number of iterations.

3.3.3 Section Search

Section Search is inspired by Golden Section Search [37] that is a way of finding a maximum or minimum for a function by sectioning. Our implementation of Section Search differ from Golden Section Search in two ways:

1. It iteratively divides the search space in half instead of using the golden ratio
2. The accuracy of the algorithm increases for each iteration, i.e. every time all muscles have been updated

Section Search works by analysing the error space with regards to one dimension, i.e. one muscle, and finds the minimum by iteratively sectioning this error function in half.

Algorithm 3 Pseudocode Section search

```
give muscles random strength and calculate error
for (each iteration)
  for (each muscle)
    select muscle by random, and get muscle strength
    STEPUP = the strength needed for maximum contraction
    STEPDOWN = the strength needed for maximum relaxation
    for (the number of times to section the search space)
      STEPUP = STEPUP / 2 and STEPDOWN = STEPDOWN / 2
      if (error by pulling muscle with STEPUP or STEPDOWN < best error)
        store muscle strength with STEPUP or STEPDOWN
        STEPUP = STEPDOWN or STEPDOWN = STEPUP
    endfor
  endfor (each muscle)
  increase the number of times to section the search space
endfor (each iteration)
```

The algorithm starts of with a random pull. Then it analyses the error graph of one muscle by sectioning the error above or below a given muscle strength in half. If one of the new muscle strengths yields a lower error value than the current position, then the algorithm pulls the muscle by the respective amount, and starts to search from there. The STEPUP or STEPDOWN value added or subtracted from the original muscle strength is used as a reference for further search. However, the value is halved every time a new stage is reached. In the first iteration, i.e. the first time all muscles are updated, the algorithm is only allowed to divides the STEP values two times. For every new iteration, the algorithm is allowed to half the STEP value once more, increasing the accuracy over time.

Figure 3.2 shows how this algorithm would search a two dimensional error function. a_1 is the initial position. In the first iteration, shown as a black line, the algorithm tries half of both extremes, and moves to b_1. The step size used to move to b_1 is used as a reference for further search. This step size is halved once more, but this time the new positions doesn't yield any lower error. As the algorithm in the first iteration only is allowed to section the search space twice, it must give control to the next muscle. In the next iteration, i.e. when all the other muscles have tried to minimize their error, the Section Search algorithm starts at a_2. This time the algorithm get to halve the step size up to three times, ending up at position b_2.

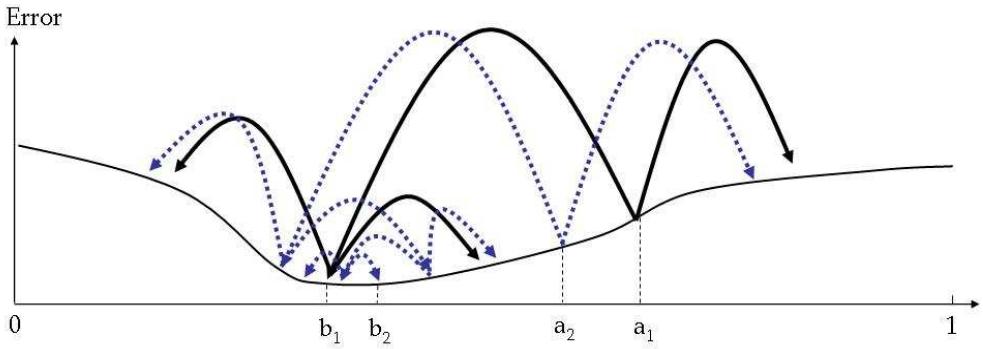


Figure 3.2: Section search during the two first iterations

3.3.4 Push or Pull

The Push or Pull algorithm is inspired by how humans might proceed to solve the problem. The algorithms is very simple. Select each muscle by random, and either contract or relax that muscle as long as the error is reduced.

Algorithm 4 Pseudocode Push or Pull

```
give muscles random strength and calculate error
for (each iteration)
  for (each muscle)
    select muscle by random, and get muscle strength
    add and subtract STEP to current muscle strength and calculate error
    if (error by contracting or relaxing the with STEP < best error so far)
      while (error is reduced)
        continue to contract or relax muscle with STEP
      store best muscle strength
  endfor (each muscle)
  STEP = STEP/2
endfor (each iteration)
```

The algorithms starts off with a random pull. It selects a muscle for update by random, and analyses the error graph by adding and subtracting a certain amount called STEP. If any of these new muscle configurations yields lower error, the algorithm sets this as the new strength, and tries to continue adding or subtracting the same STEP value as long as the error decreases. If none of the new positions yields a lower error, the algorithm gives control to the next muscle. For each iteration, i.e. every time all muscles have been adjusted, the STEP value is halved. The STEP value is set to 0.5 for the first iteration.

Figure 3.3 show how this algorithm would descend a two dimensional error space. In the first iteration, the algorithm starts off at the initial position, a_1. From here, it adds or subtracts 0.5. None of the new positions gives a lower error, so the algorithm gives control to the next muscle. In the second iteration, the algorithm starts off at a_2. This time the step size is 0.25, and the algorithm finds lower error at b_2, and moves there. Being successful, the algorithm tries to subtract 0.25 once more, but without success. It stays at b_2, and gives control to the next muscle.

To summarize, the main difference between Section Search and Push or Pull is that Section Search always start off from each extreme value and gradually get to search with more accuracy. Push or Pull on the other hand uses a step value that decreases for every iteration, but in each iteration it gets to use the same step value as many times as desired, as long as the error is reduced.

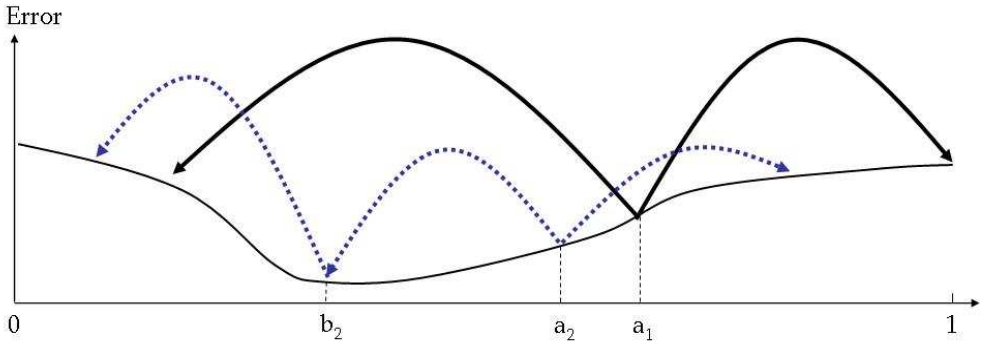


Figure 3.3: Push or pull during the two first iterations

3.3.5 Genetic Algorithms

Genetic Algorithms (GA) [8, 16, 32, 27] uses random search and methods inspired from biology to find the global minimum of an error space. The idea behind Genetic Algorithms is to use the analogy of nature to solve complex real world problems. Its strength lies in its general usefulness, and not because it is optimized for some specific problem. GA can therefore often be outperformed by more specialized schemes tuned for the task at hand.

Coding a problem with GA

To solve a problem with Genetic Algorithms, one first has to derive a way to write the solution to the problem as a finite length string. Normally a binary string is used, as computers works on binary numbers. As an example, suppose we want to maximise the optimization problem of figure 3.4. In other words, we wish to maximize the function $f(x) = x^2$ in the interval $[0, 31]$, and come up with $x = 31$. By coding x as binary number with 5 digits, we enable x to take on all values from $0 - 31$ in decimal value. Such a string would be called an individual in GA terms, and would be analogous to the human genome in the real world. While the human genome is built up of DNA molecules consisting of the four bases A,G,T and C, the binary string is made up of only 0's and 1's. Several such strings or individuals will form a generation. The task for a Genetic Algorithm will therefore be to create the individual 11111 (decimal 31) that would maximize the output of the function in our range. Genetic Algorithms achieve this by performing natural selection amongst the individuals from one generation to the other. In this way, a larger proportion of the best individuals from one generation are reproduced to the next, ensuring the survival of the best genes.

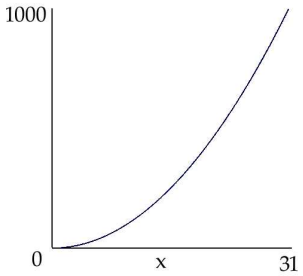


Figure 3.4: The x^2 function in the range [0,31]

Pseudocode

To solve the problem depicted in figure 3.4, a standard Genetic Algorithm would work this way:

Algorithm 5 Pseudocode GA

1: create a random initial population (binary strings with five digits)
2: use natural selection to find the best individuals in the generation
3: create a temporary generation by reproducing the best individuals
4: use cross over and mutation on this temporary generation to create new generation
5: repeat from point 2 until the desired individual (11111) is born

Create a random population

Suppose we create an initial population of four individuals. Lets assume that these individuals are made by random, and have the following values shown in table 3.1. Of

these four individuals, we have to select which ones to reproduce, and which ones to mate, in order to create a new generation. To accomplish this, we have to define a fitness value. One obvious and useful measure would be the output of our function $f(x) = x^2$. Table 3.1 shows each individuals fitness based on this definition. The relative fitness is each individuals fitness relative to the fitness of the whole generation, and is shown in the last column of table 3.1.

Individual	Binary value	Decimal value	Fitness value	Relative fitness
1	01101	13	169	14.4%
2	11000	24	576	49.2%
3	01000	8	64	5.4%
4	10011	19	361	30.8%
		Sum	1170	100%

Table 3.1: Initial population

Natural selection

If we were to reproduce an individual from this population, we say that there is 14.4% chance that we will pick individual 1. For individual number 2, the chance is 49.2%, and so fourth. We pick the individuals by random based on their relative fitness, and place the ones selected for mating in a mating pool, analogous to natural selection in biology. Assume we did this four times for our example, and came up with individuals 4,2,1,2. The mating pool would then consist of:

Individual	Binary value	Decimal value
1	10011	19
2	11000	24
3	01101	13
4	11000	24

Table 3.2: Mating pool

Cross over

Mating of these individuals would normally be a two stage process, consisting of selecting the individuals to mate, and then applying cross over to each pair of individuals. Cross over is a mechanism from biology where the chromosomes line up in meiosis and exchange genes[4]. Using random we select mating partner and cross over point between the individuals. Figure 3.5 shows a cross over between individual 1 and 4 at the third binary cipher, giving us two completely new individuals each with a different fitness value. Table 3.3 shows the new population given cross over point at position 3 for individual 1,4 and position 1 for individual 2,3. We can see that the generations total fitness has increased from the previous generation by 720, from 1170 to 1890. We also find that the fitness value of the fittest individual in the generation has increased by 256, from 576 to 841. As we can see, the Genetic Algorithm have used the stronger genes to breed a new and better generation, with individuals closer to our target than the previous generation. Repeating this process for some more generations would probably lead the Genetic Algorithm into creating the individual (11111), which would maximize the function drawn in figure 3.4, thereby solving the problem presented.

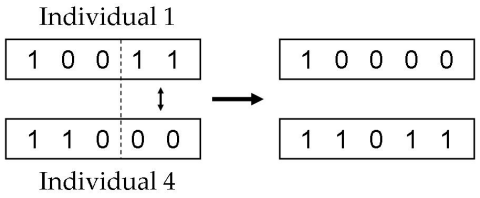


Figure 3.5: Cross over between individual 1 and 4 at the third binary cipher

Parent	Individual	Binary value	Decimal value	Fitness value	Relative fitness
1,4	1	10000	16	256	13.5%
2,3	2	11101	29	841	44.5%
2,3	3	01000	8	64	3.4%
1.4	4	11011	27	729	38.6%
			Sum	1890	100%

Table 3.3: The new population given cross over point at position 3 for individual 1,4 and position 1 for individual 2,3

Mutation

In addition to natural selection and cross over, Genetic Algorithms are based on mutation. Mutation is in biology nothing more than error in the replication process of DNA, introducing a wrong base, or inserting or deleting an extra base in the DNA. Translated into Genetic Algorithms, mutation becomes the inversion of one or more binary ciphers. The reason why mutation is used can easily be visualized by looking at a more complex problem than maximizing x^2 function. Figure 3.6 shows a more complex function in the interval [0,31]. If all initial individuals were created in the range [0,22], the Genetic Algorithm would select, replicate and cross over for ever, but never be able to maximize the function. The highest achievable fitness number would be 450, and not 1000 as desired. Introducing mutation, enables the creation of an individual with a higher number than 22, thus removing the danger of letting the Genetic Algorithm stagnate in a local maximum.

Using GA in the simulator

In the context of our simulator, each face expression, i.e. a specific muscle force applied to each muscle, is regarded as an individual. Each individual is coded by a binary string that represents the specific contraction for all muscles. If a muscle is coded with 7 binary digits, then a face with two muscles will have individuals of 14 bits length. For each generation the best individuals are reproduced according to their fitness, i.e. the muscle configuration that yields least error. To ensure that the best genes are preserved between generations, we have introduced elitism. Elitism makes sure that the best individuals in each generation are moved on to the next generation regardless of whether they are selected for reproduction or not.

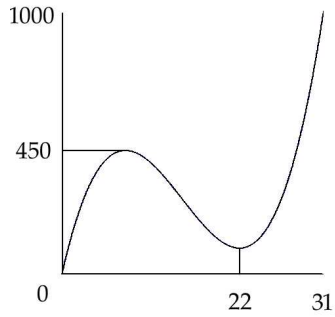


Figure 3.6: A function showing the need for mutation. If the original generation only contains individuals in the range [0,22], GA will never be able to come up with the target individual 31 without mutation

3.4 The experiment

In order to test the performance of each algorithm, they were exposed to three different facial structures with four different target positions. Each structure's composition is inspired by anatomical structures in the human skin, though in a very simplified manner. For each target position, each algorithm got to pull the muscles up to 100 times in order to come up with the desired face expression. To make the challenge equal for all algorithms, they all had to make the first pull by random.

Skin resolution and accuracy

The resolution of the skin was set to $600 * 600$ pixels. A muscle pulling a single dot the X-axis direction could therefore in theory give this atom up to 600 different X-axis positions. However, the length of most muscles did not cover more than half skin area, and they were all given an upper bound on the maximum contraction. The accuracy of each muscle therefore typically became between $0.2 - 1\%$. Less contraction would not move a point to a new pixel, and the error would thus remain constant.

Noise

As described in chapter 2, noise in the simulator can be introduced from two different sources. The noise from each muscle was set to 70%, i.e. there was 70% chance for a muscle to be affected by noise. The size on this noise however was quite small, and was bound to a maximum of 2.5% in each direction from the desired muscle force. Depending on the size of the muscle, this typically would create an error in the range of $0 - 4$ pixels in each direction. Noise from the feedback however was set to be more severe, but was only introduced occasionally. 5% of the face expressions were affected by feedback noise, and the size could be up to 2% of the face resolution. This implies that feedback noise gave between $0 - 12$ pixels in error for each direction. Figure 2.6(b) shows the size of the noise added in a typical error landscape.

Parameters

For each algorithm, there are several parameters that can be adjusted to tweak their performance. Nevertheless, in order to get an impression of their general ability to solve the problems at hand, the parameters were set so that each algorithm would perform fairly well under all conditions. We therefore allowed no adjustment of any parameters between each target state and face configuration. Hill Search got an initial step value of 1, while the Push or Pull got a step value of 0.5. The Section Search had to use half the muscle strength from its current position to 1 and 0 as step values for contraction and relaxation respectively. The Genetic Algorithm created 10 individuals and 10 generations. The best overall mutation rate was found to be 20%, while the number of cross over points was set to three. Seven bits was used to represent each muscles strength, enabling up to 128 strength alternatives per muscle.

3.5 How to interpret the results

To get reliable results, each algorithm was run 1000 times, and the values for each run were averaged.

As mentioned in chapter 2, the initial configuration, i.e. when all muscles are relaxed has the error value 1. The error at the target position is defined as zero. The worst obtainable error for an algorithm after 100 iterations can therefore only be 1. As all algorithm started off with a random pull, their initial average error should all be the same. Every time an algorithm created a new muscle configuration which reduced the error, the new result was stored in memory. If a new muscle configuration created a face expression with higher error, then the best error so far was stored. In this way all graphs became monotonically descending. The algorithm that produces the steepest average decent towards zero or reaches the lowest error in fewest amount of trials are the best performing algorithm for the task at hand.

To see the size of the deviation in our results, we repeated the process three times for each algorithm and target facial expression. However, the deviations were so small that they didn't affect the conclusion, but we have omitted drawing all the runs as they tend to disturb the graphs.

Using Stochastic Search as a performance measure

As we mentioned in section 3.3, Stochastic Search can be used to benchmark other algorithms. However, we can also use Stochastic Search to say something about the topology of the error space. If Stochastic Search on average ends with a low error in a given example, the proportion of the error space with low error has to be bigger than for the others targets. Using simple statistics, we can calculate the proportion of the error space that is below a certain average error.

A simplified illustration

To understand this, let us take a simplified example of a box full of white and black balls. Imagine that we stick our hand down in the box with our eyes closed, and pull out a ball. After picking up the ball, we write down its colour before putting it back into the box again. Suppose now that there are only 10% white balls, and 90% black balls. How many balls should we pull out before having pulled up a white ball with more than 95% probability? Lets denote the probability for pulling a white ball $P(w)$. Let $P(s)$ denotes the probability for success which we want to be more than 95%. The formula for calculating the number balls we have to pull out then becomes:

$$P(s) < (1 - P(\bar{w})^x) \tag{3.1}$$

Where x is the number of balls we lift up, and $P(\bar{w})$ is the probability for picking a black ball. This equation can easily be solved with respect to x:

$$x > \frac{ln(P(\bar{s}))}{ln(P(w))} \tag{3.2}$$

Inserting for 0.1 for $P(w)$ and 0.95 for $P(s)$ gives $x = 28.43$. In other words, if we pull the 29 balls, there is more than 95% chance that we have lifted up a white ball from the box.

Average error in regions of lowest error

Relating this to the performance of Stochastic Search, we see that after 29 tries, Stochastic Search has with more than 95% probability tried out all regions that comprises less that 10% of the total error space. We call this the resolution of Stochastic Search. Note that if the error descends to 0.2 after 29 trials, the average error in the region with lowest error comprising less than 10% of the entire space is 0.2. How large

part of this region that has an error of less than 0.2 is not possible to tell, as this depends on the topology of the error region in focus. The equation 3.1 can be solved with respect to $P(w)$ and give us the resolution after x trials. This gives us:

- After 30 trials the resolution is 10%
- After 50 trials the resolution is 6%
- After 100 trials the resolution is 3%

Having these numbers in mind is interesting when we look at the performance Stochastic Search throughout the different target expressions. It gives us a clue about the average error of the lowest 10%, 6% and 3% of the error landscape, when this is impossible to visualize in three dimensions.

Size of the error landscape

It is also important to realize that the size of the error space increases exponentially with the number of muscles. If a muscle have 100 different strength alternatives, the size of an error space with four muscles are 10.000 times larger than an error space with two muscles. The results can easily been seen on Stochastic Search. Figure 3.7 shows the how Stochastic Search would descend three different error spaces ranging from one to zero, with linear functions in all dimensions. The error spaces are of 3, 5 and 7 dimensions respectively. Obviously, Stochastic Search start off with an error of 0.5. For three dimensions the error after 100 iterations is 0.06, while it is 0.17 and 0.22 for 5 and 7 dimensions respectively. These figures can also be interesting to have in mind when we look at the graphs. They can tell us something about the complexity of the error landscapes traversed by our algorithms compared to an error space only consisting of linear functions.

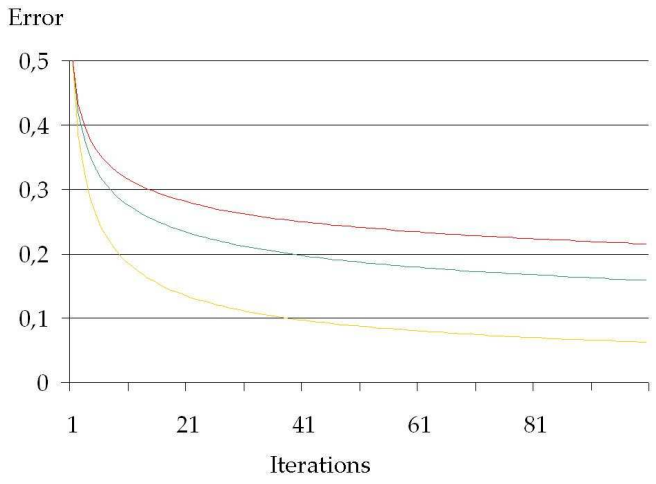


Figure 3.7: Average error obtained by Stochastic Search after 100 iterations in an error space ranging from 1 to 0 having only linear functions. The error is along the vertical axis, while the iterations is along the vertical axis. The yellow, green and red lines are for spaces of 3,5 and 7 dimensions respectively.

3.6 Results of searching the simulator

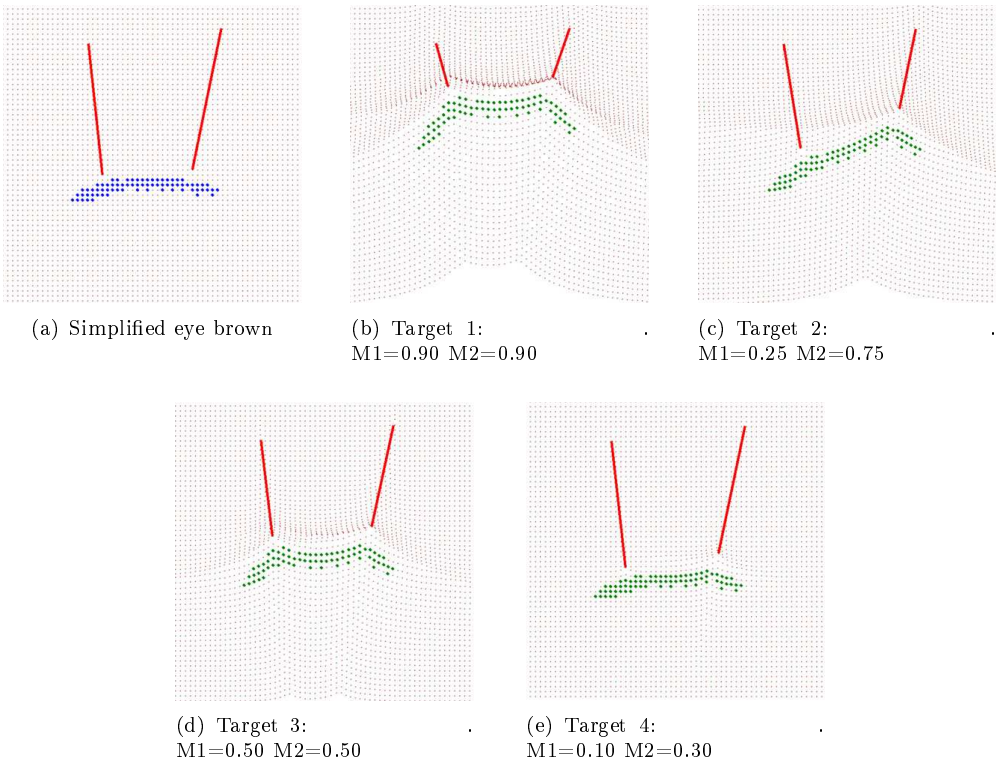
For the following discussion, it is important to emphasise that we have created three different facial structures which we often refer to as:

- Structure 1 (A simplified eye brown)
- Structure 2 (A straight wrinkle)
- Structure 3 (Two wrinkles with opposing muscles)

These structures must not be confused with their four respective target expressions labeled Target 1 to 4 which are the goal states the algorithms are trying to reach.

3.6.1 Structure 1 (A simplified eye brown)

The first structure is a simplified eye brown with two muscles and 71 points. This face configuration was chosen because two muscles enable the creation of a three dimensional error space, making is possible to visualize the challenge each algorithm is exposed to. In addition, we can see the size and effects of the noise added in the simulator, and what kind of impact this has on the error space and algorithms. Figure 3.8 show the simplified eye brown and its target positions. The initial position of the structure is shown in blue, while the target positions are shown in green.



(a) Simplified eye brown

(b) Target 1: M1=0.90 M2=0.90

(c) Target 2: M1=0.25 M2=0.75

(d) Target 3: M1=0.50 M2=0.50

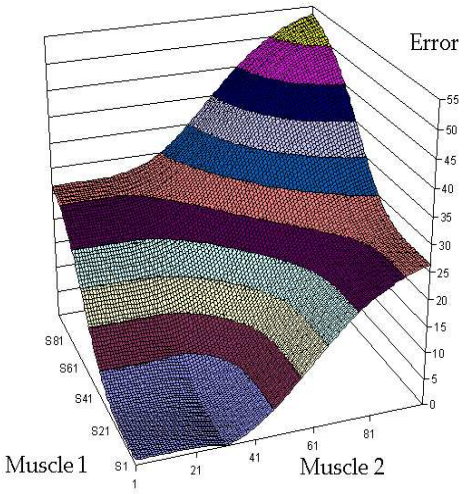
(e) Target 4: M1=0.10 M2=0.30

Figure 3.8: Structure 1, and its four target positions and their muscle strengths

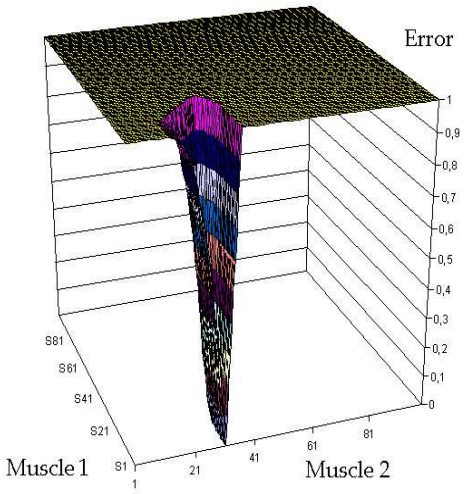
The error landscapes

Even though the four target expression look quite similar, the way the simulator is constructed and the way error is measured, creates error landscapes with very different

characteristics. Figure 3.9(a) shows the error space of Target 4 without noise, and illustrates the effects the error measure and the simulator has on the algorithms. At first glance this error space looks quite easy to traverse, but notice that the maximum error achieved in this error space becomes close to 55. This means that if both muscles are pulled with maximum strength, the sum of distances travelled by the points become almost 55 times as big as for the initial position. Stated otherwise, the target state to be reached is very close to the initial position, as can be seen in figure 3.8(e). To see the effect this has on the algorithms, we take a closer look at the region in this error space where the error is below one, shown in figure 3.9(b). Even though this is the same error space, the challenge has become a totally different one. It is only after the algorithms have fallen into this pit that they become productive and can make corrections to the face.



(a) The entire error space of target 3



(b) The error space of target 3 below 1 in error

Figure 3.9: The error space of Target 4. As for all the error spaces in this section, each muscle's strength in % can be read out from the two axis spanning the floor of the error space, while the resulting error is along the vertical axis

Target 1

Going back to the different target expression, we start of by looking at simplest achievable target expression with respect to Stochastic Search, namely Target 1. Figure 3.10 shows the error space, and the results obtained by the algorithms.

Without noise

Stochastic Search obtains on average an error of 0.015 after 100 runs, indicating that the area of low error is bigger than for a pure linear error space. Looking at the curves and the error space in figure 3.10 confirms our theories. We see that from the initial position (both muscles at zero strength) the error space is continuously descending in all directions, all the way down to the global minimum. From the global minimum there is a little increase towards the end of the error space, explaining why Stochastic Search obtains lower error than for a linear error space. Knowing that the target muscle strength is 90% for each muscle explains the shape and symmetry of the landscape. The initial start error is close to 60%. This is easily understood by

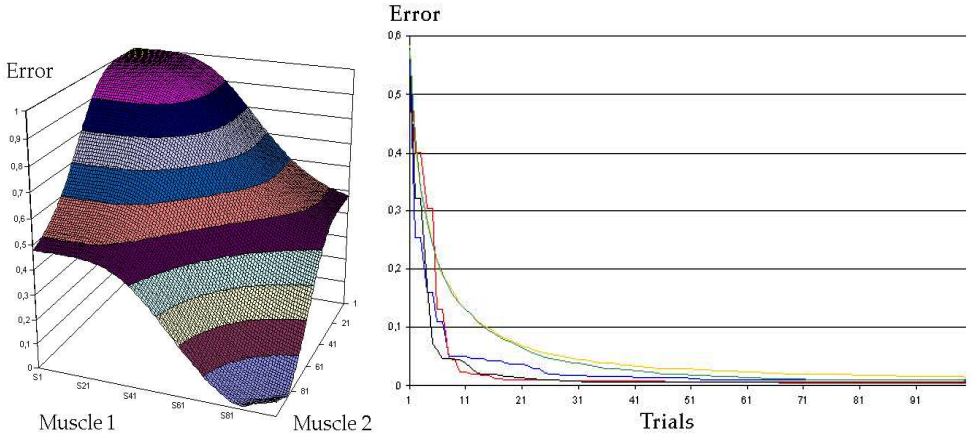


Figure 3.10: Target 1 without noise. The error space is to the left, and the performance of each algorithms to the right. As for all following graphs in this section, Stochastic search is shown i yellow, GA in green, Section Search in red, Hill Search in blue, and Push or Pull in black. The error is along the vertical axis, while the number of online trials is along the horizontal axis

realizing that error landscape is close to linear, with approximately half of the muscle configurations having an error less than 0.5.

Looking at each algorithms performance we see that both Stochastic Search and GA have close to the same performance, and the same goes for the three other algorithms. However, GA performs a little better than pure Stochastic Search. This is of course due to the fact that GA to some extent preserves the best individuals, and uses its evolutionary techniques to achieve faster convergence. Nevertheless, in this error landscape, these techniques don't pay off that much.

Looking at such a uniform fitness landscape, one could believe that the Hill Search would perform better than Section Search and Push or Pull. The reason this is not the case is due to the way our Hill Search is implemented. While the two other algorithms checks one dimension at a time, the Hill Search check both before making a move. Nevertheless, the Hill Search does not check all combinations of the two dimensions, and will therefore have to move in the same zigzag pattern as the two others. Apparently, in an error space shaped like the one in figure 3.10, this extra overhead in finding the best dimension to descend the error space do no increase in efficiency compared to the extra overhead induced by the method.

With noise

The next question becomes how noise affects the performance. Figure 3.11 shows what the size of noise added does to the error landscape.

It is quite interesting to see that the size of the noise affects this error landscape to a much lesser degree than for the other examples, e.g. figure 2.6(b). This is of course due to the way error is defined. An android trying to correct a facial expression that is disturbed by a little amount doesn't produce that much error if the target facial expression is far from its initial position. On the other hand, the same amount of noise can have severe impact if the android is to perform small facial corrections.

Looking at the curves in figure 3.11 we see that the overall impact from noise is so small that it can almost be neglected. The Stochastic Search is obviously not affected by the noise as it uses no feedback at all. The Genetic Algorithm is to a large extent based on random search, and will therefore not be that much affected either. The three remaining algorithms on the other hand should be affected by the noise. The reason this noise is not that devastating is due to the fact that the noise is random. If

noise makes the algorithm perform a wrong decision, the algorithm could move back in the next iteration given correct feedback.

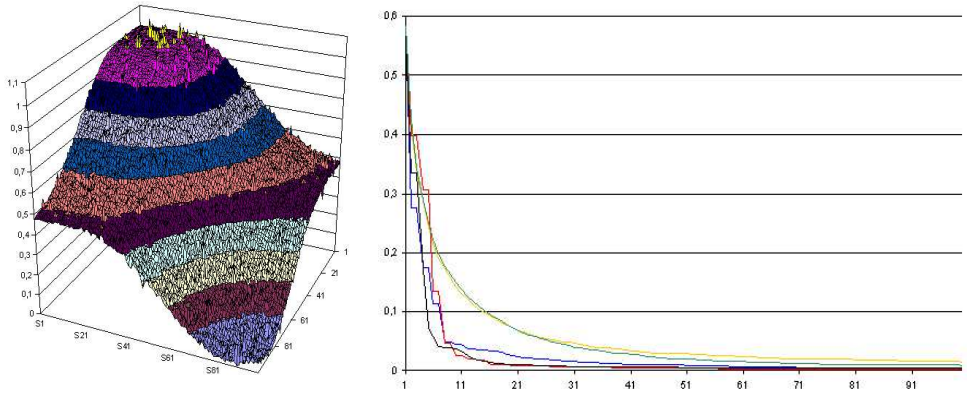


Figure 3.11: Target 1 with noise

Target 2

Using the result of Stochastic Search, we see that the second easiest facial expression is Target 2. The target muscle strength is 25 and 75%, and the resulting error space without noise is shown in figure 3.12.

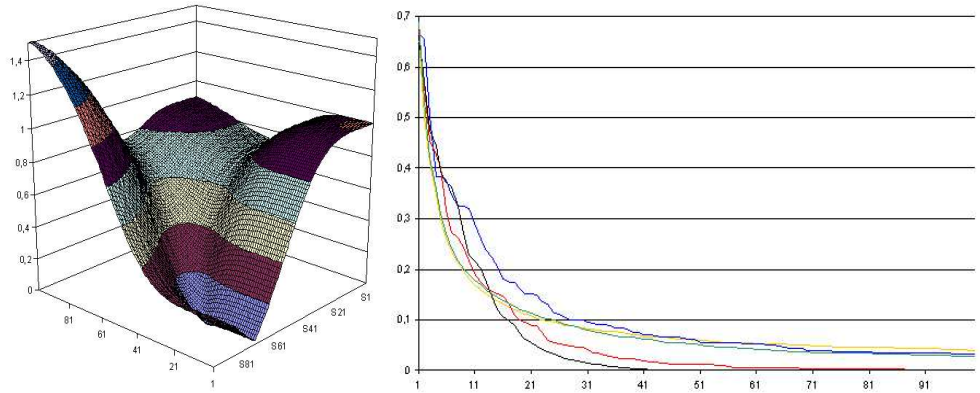


Figure 3.12: Target 2 without noise

Without noise

Stochastic Search on average ends up with an average error of 0.04, and achieves an average error of approximately 8% within the best 10% of the error landscape, clearly indicating that this is a more difficult problem to solve than Target 1. In addition there are two plateaus, and two valleys present that can induce problems for the algorithms.

Looking at the results, we see that the performances of the error based algorithms are reduced compared to Target 1. This can be a result of the steepness of the error space which has increased from Target 1. The error based algorithms have to reduce their step value before descending the error landscape efficiently. This is most apparent by looking at Hill Search, which has the most dramatic reduction of performance compared to the random based methods. Nonetheless, the Push or Pull algorithm tends to give slightly better results.

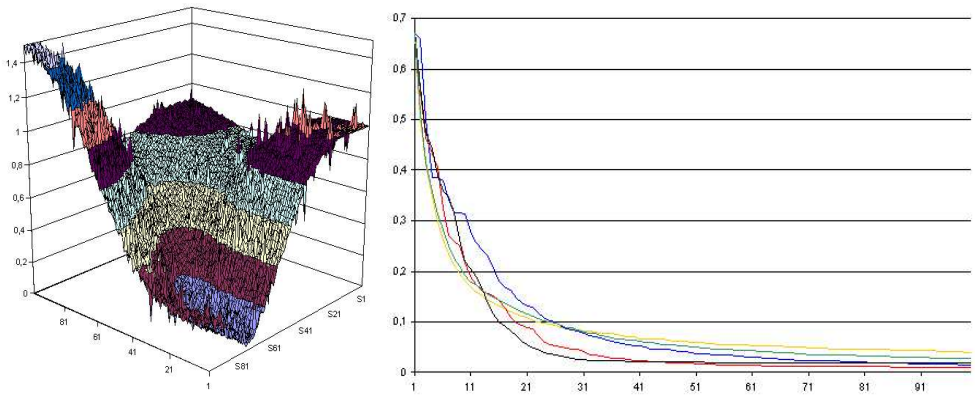


Figure 3.13: Target 2 with noise

With noise

Figure 3.13 shows how the noise affects the error landscape. As we can see, the size of the noise has increased from figure 3.11. This is because the target position is closer to the initial position, and the need for accuracy increases. Nevertheless, figure 3.13 shows that the noise does very little harm to the algorithms, due to the fact that it is random, and is still not too big to disrupt the original shape of the error landscape.

The only algorithm affected by the noise is the Push or Pull algorithm. Instead of ending up at zero error, it converges to 0.018. The reason for this is clearer in other examples, but the effect can be seen here already. Notice that Push or Pull flattens out around 50 tries. At this stage the step value can have become fairly small. The size of noise close to the global minimum can make the algorithm make a lot of mistaken moves, further reducing its step value. In a worst case scenario, the step value can become less than 0.1% of the muscle strength. Step values of this size will yield a muscle movement that is smaller than the resolution of the skin, preventing the algorithm from performing any more corrections.

Target 3

Moving on to the second most difficult target expression, we see that Stochastic Search only obtains 13% of the target value, telling us that the average error in the best 3% of the error landscape is about 0.13. These values tell us that the global minimum is quite narrow, and that there is a steep gradient towards this global minimum. Figure 3.14 shows this error landscape without noise.

Without noise

It is interesting to see the steepness of the error space descending toward the global minimum. This has approximately the same gradient in all directions, and its second derivative is always negative. Looking back at figure 2.4 which shows the non linear muscle activation function explains this phenomenon. The value of each muscle in this target expression is 50%, and small discrepancies from this value yields large fluctuations in each muscles output. This explains why the second derivate of the error space are negative in all directions.

Looking at the performance of each algorithm, this is the first target where there is a superior algorithm, namely Push or Pull. Already after 40 iterations, the error is below 0.01. In contrast, Stochastic Search have reduced the error with 80%, having an error of 0.19 after 40 tries.

GA and Stochastic Search have close to the same performance, while Section Search and Hill Search ends up in between Push or Pull and the stochastic methods. We are

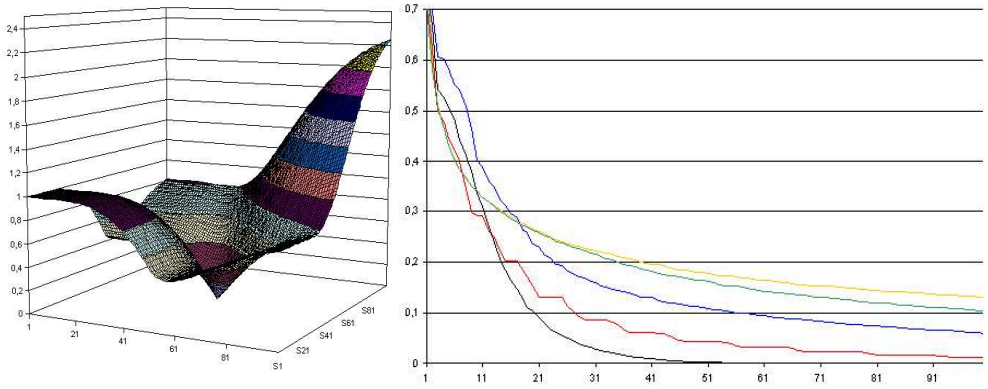


Figure 3.14: Target 3 without noise

starting to see the pattern that the slower dampening factor of Hill Search, and the mechanisms for avoiding Section Search to get stuck are at a cost, namely slower decent of the error landscape.

With noise

Introducing noise reduces the efficiency of the best performing algorithm, while the others are more or less unaffected. Figure 3.15 shows the correlation. Looking at the error space with and without noise from the error of 0.2 to zero, we see that noise widens the pitfall. In effect, this fools the Push or Pull algorithm to believe it has obtained low error when it hasn't, getting it stuck outside the original pitfall. Figure 3.16 shows the pit with and without noise.

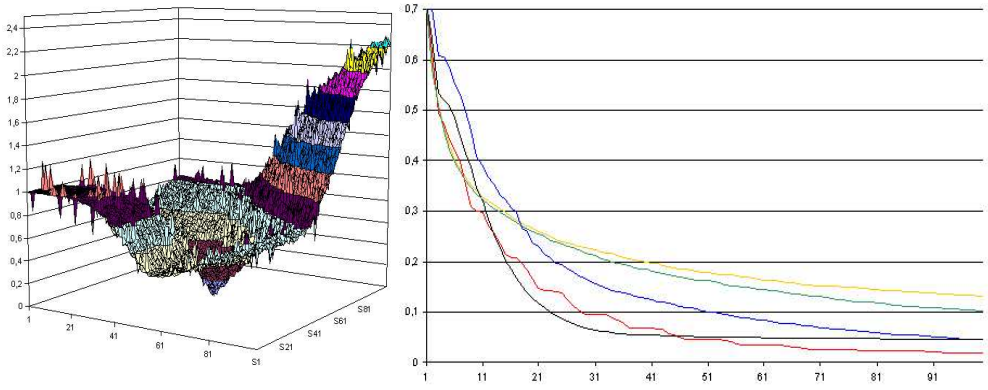
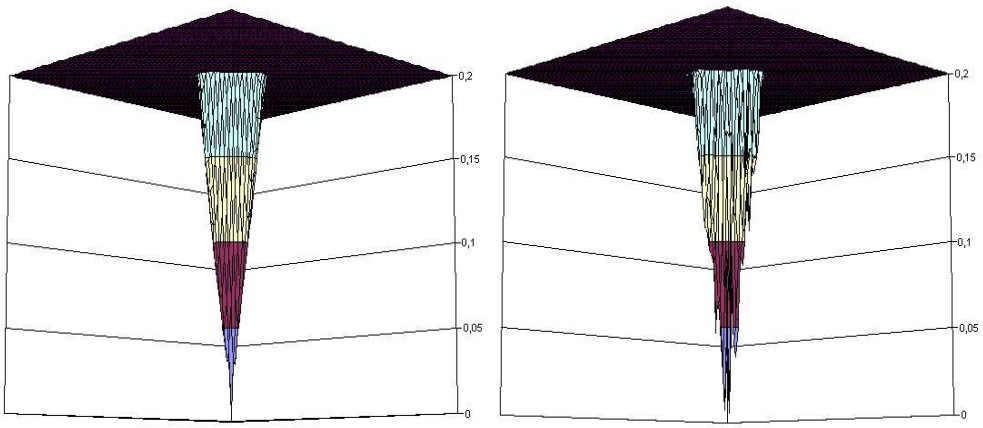


Figure 3.15: Target 3 with noise



(a) Without noise (b) With noise

Figure 3.16: Target 3 below 0.2 in error

Target 4

As mentioned above, the seemingly nice error landscape of Target 4 becomes the most difficult to traverse when the error is regarded from one to zero. The area where error is below one is definitely the smallest of the error spaces created by our first facial structure. This is not so strange when we know that the target strengths are 10 and 30%, and that the distance the points can be moved is up to 55 times the initial distance. Only small muscle strengths around 10 and 30 % give lower error than one, while the rest yields substantially higher errors.

Looking at the shape of the pitfall, on realize that even though an algorithm falls into it, the width of the pitfall is very small compared to its length, making correct manoeuvres in this deep valley hard to perform. Stochastic Search tells us that the average error is close to 0.5 for the best 6% of the error space, giving us a clue about the size of the pit with error below 1. Looking at the error curves in figure 3.17 without noise, we get to see some interesting differences.

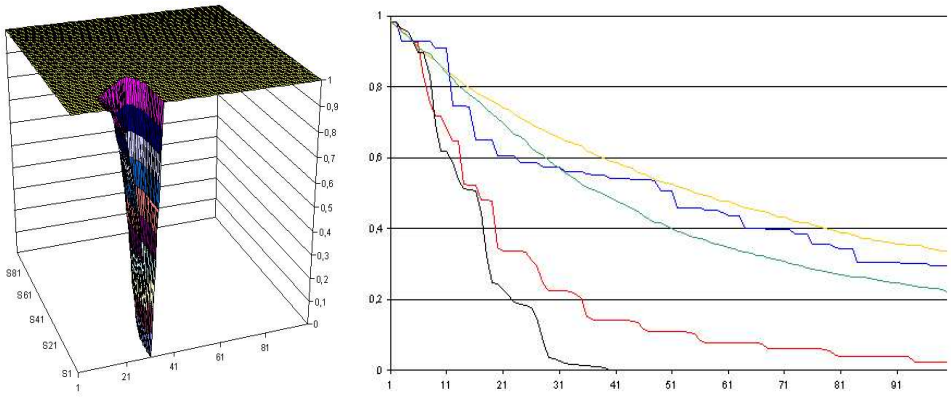


Figure 3.17: Target 4 without noise

Without noise

Again we see that Push or Pull is the winning algorithm. But this time the efficiency of Hill Search have become severely reduced, while GA has improved. There are several different reasons for this behaviour.

The main reason why GA has improved is due to the elitism model, and the way the best individuals are selected for reproduction. The Genetic Algorithm is based on fitness and not error. The way fitness is defined sets some limits to how the error landscape looks like for GA. In contrast to the three other algorithms, GA looks at the error space exactly the same way as figure 3.17, only upside down.

The difference between defining a fitness value and an error space is the definition of zero. While zero error is easily defined when dealing with error spaces, zero fitness can be harder to define. Usually one does not know size of the size of the entire fitness space, and thus a useful measure could be the initial start position of the error space to be searched. If the start position is defined as zero fitness, then all muscle configurations that give lower error, will result in higher fitness, but error spaces with higher error will only yield zero fitness. Hence the fitness landscape traversed by GA would look like the one in figure 3.18.

GA uses random generation of individuals until individuals with higher fitness than zero are created. For this reason GA and Stochastic Search will normally descend the error space exactly the same way initially, but once GA creates an individual that rises from the flat surface, all individuals in the next generation will be offspring from this individual. The crossover mechanism will thereafter try to climb the hill. GA therefore performs much better than pure stochastic in this example due to the fact that after a little while, only muscle configurations that yield an error in this narrow pitfall will be reproduced.

The reduced performance of Hill Search has its own reason. Looking at figure 3.17, we see that the pitfall is very narrow. Knowing that the Hill Search algorithm has a hyperbolic dampening factor, we realize that it uses some time before the step value is small enough to make it fall all the way down the pitfall. At the 100th iteration, the step value of Hill Search is $1/25 = 4\%$, which is almost the width of the pitfall around 30% error, explaining the high error.

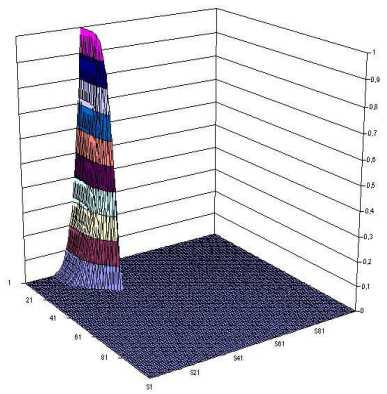


Figure 3.18: Fitness of Target 4 without noise. This is the way the Genetic Algorithm sees the error space

With noise

Turning on the noise have a larger impact on the algorithms than for any of the other target positions for this facial structure. This is because the size of the noise added becomes considerable in the region of interest. Looking at the region where the error is below 1 shown in figure 3.19 gives at totally different landscape. The pitfall has become heavily corrupted with high peaks and pitfalls on each side.

As we see, the noise has affected the performance of the Hill Search algorithm substantially. This is of because the induced error has widened the pitfall, allowing the algorithm to move to positions outside the pitfall, that it wouldn't have reached else wise. These new positions yields better error value in the next round as the same step value makes it fall deeper into the valley. The same reason might explain why GA has improved a little bit. Values close to the global minimum that normally would have substantial error could now by random have strongly reduced error, selecting more of these individuals for selection, which all are close to the goals state.

We also see that the performance of all three error based algorithms are smoothed due to the noise added. This just tells us that there is a wider spread in the descending patterns towards the global minimum point. As before, the noise added makes the Push or Pull algorithm suffer form increasingly the small step sizes, stagnating after approximately 50 iterations.

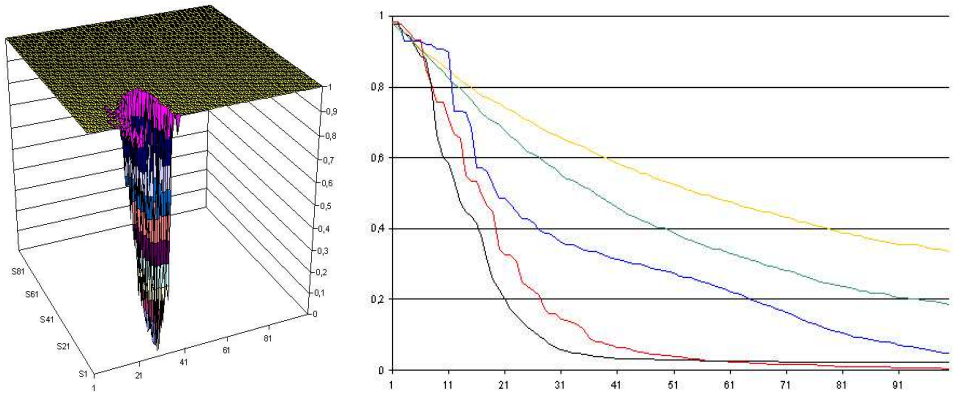
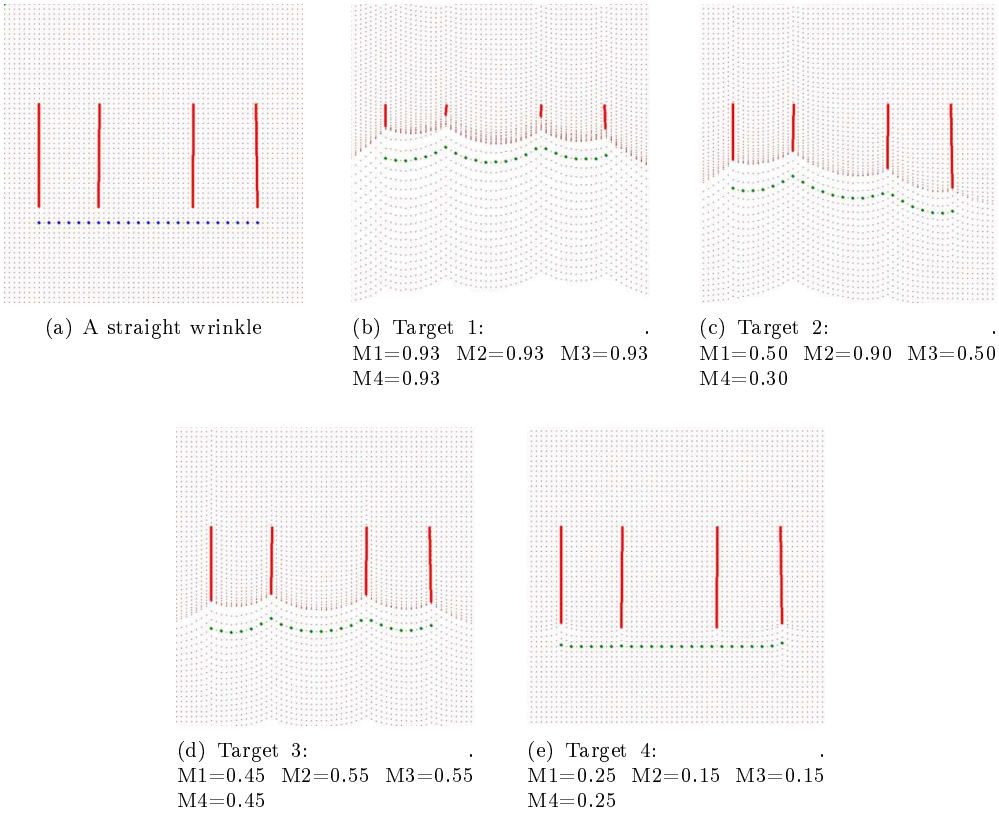


Figure 3.19: Target 4 with noise

3.6.2 Structure 2 (A straight wrinkle)

The second face structure can be looked upon as a very simplified straight wrinkle. The facial structure and its target positions are shown in figure 3.20. This structure has got four muscles instead of two, and the error space cannot therefore be visualized any longer. As a consequence the size of the error space has also increased. Even though it is no longer possible to visualize the error space, our theories from looking at facial Structure 1 should also apply here.



(a) A straight wrinkle

(b) Target 1: M1=0.93 M2=0.93 M3=0.93 M4=0.93

(c) Target 2: M1=0.50 M2=0.90 M3=0.50 M4=0.30

(d) Target 3: M1=0.45 M2=0.55 M3=0.55 M4=0.45

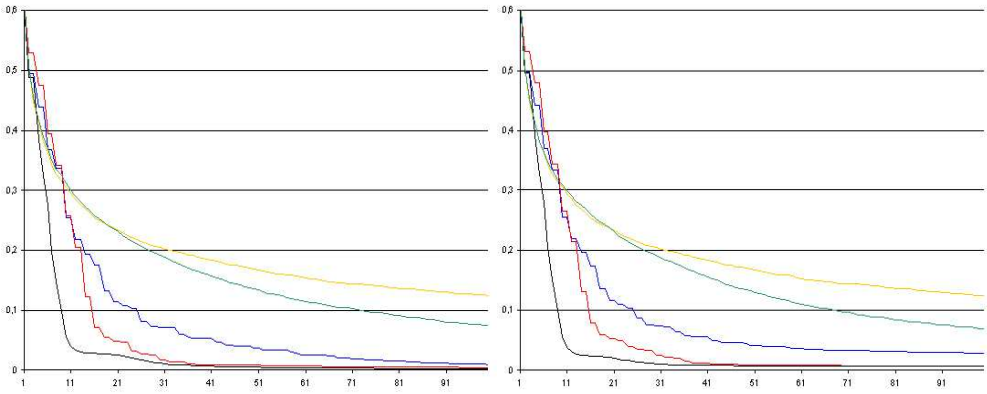
(e) Target 4: M1=0.25 M2=0.15 M3=0.15 M4=0.25

Figure 3.20: Structure 2, and its four target positions and their muscle strengths

Target 1

The first target expression is inspired by Target 1 in face Structure 1. Each muscle is pulled by 93%, and therefore almost all muscle configurations should yield a reduced error compared to the start position. Figure 3.21 shows the curves with and without noise.

As for the simplest target in Structure 1, the first pull gives an error about 60%. The algorithms shows the same pattern in performance, but their performance is more spread, which can be a result of the increased space to search. As with Target 1 in Structure 1, noise does not affect the algorithms by much, except from Hill Search that experience a little loss in performance.

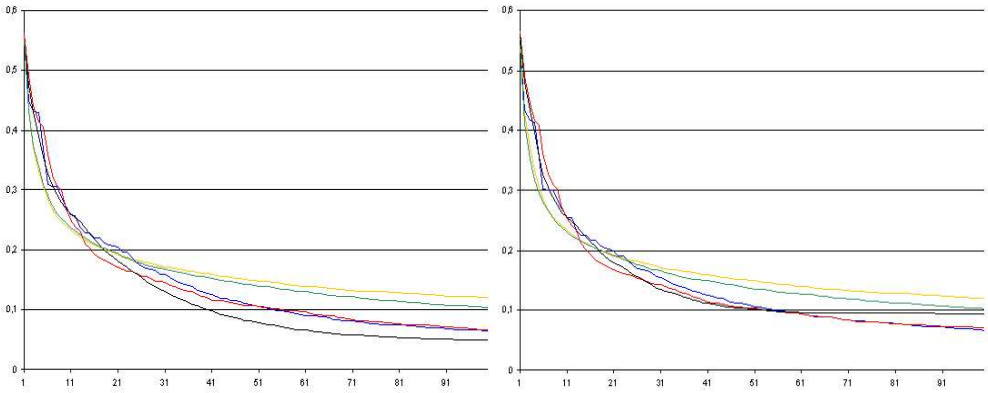


(a) Without noise (b) With noise

Figure 3.21: Target 1

Target 2

Even though the muscle configuration is quite different from Target 1, the resulting error space shows some of the same characteristics. Knowing that the muscle strengths are: 50, 90, 50, and 30%, we realize that small muscle forces should give face expressions with approximately the same distance from the target value as large muscle contractions. This correlates well with the fact that the first random pull has an error of approximately 55%. The curves of the algorithms with and without noise are shown in figure 3.22.



(a) Without noise (b) With noise

Figure 3.22: Target 2

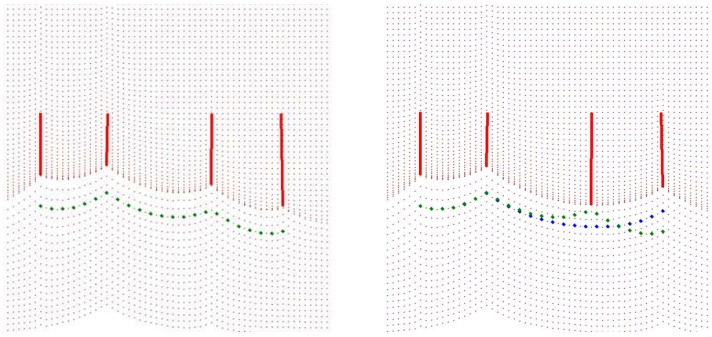
Plateau

We see that noise has very little impact on the algorithms, except from the Push or Pull algorithm. Looking at its performance without noise, it seems like its stagnating around 5%. One can be tempted to believe it has reached some form of plateau where all surrounding points have the same or higher value. Looking closer at figure 3.23 might explain this.

If the strength of muscle three is reduced while strength for of muscle four is

increased, the resulting positions might give close to the same error. By holding the two first muscles constant at their target value, we can produce an error space for muscle three and four, shown in figure 3.24.

As we see, this error space has got a flat area with low error, and a valley with even lower error. The plateau has an error of 18% and the valley around 3% error. Both these areas are very difficult to traverse for the Push or Pull algorithm if the step size has become too small to search outside these flat regions. If a flat or close to flat area is affected by noise, the algorithm can be exposed to the same problem as for Structure 1. The high presence of error in the flat area will force the algorithm to use smaller step values, ending up becoming too small for the resolution in the simulator. As this occurs around the 60th iteration, the algorithm can in a worst case scenario have reached a level where the step size is below 0.08% of the muscle strength, creating problems for the simulator.



(a) Target 2

(b) Muscle configuration yielding "flat" error area

Figure 3.23: Target 2 is to the right. To the left: An alternative muscle configuration that have close to the same error for several muscle strengths of muscle 3 and 4

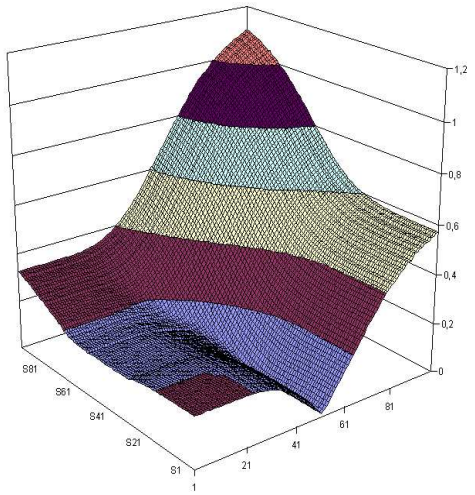
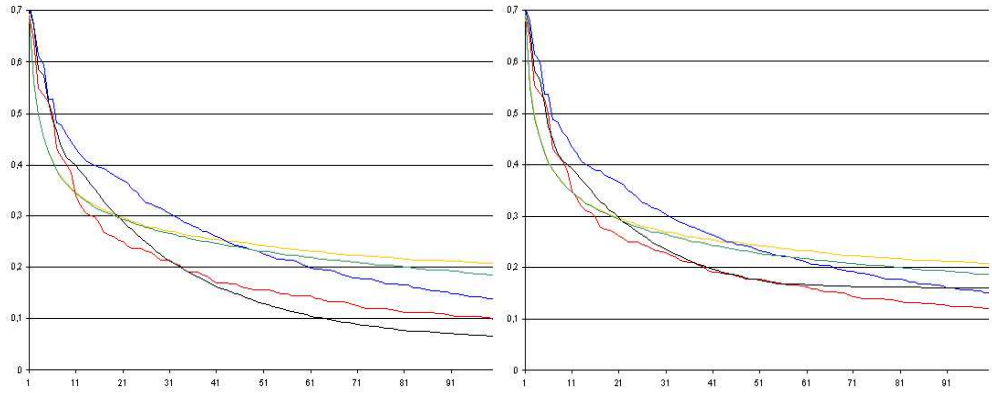


Figure 3.24: Error space for muscle 3 and 4 in Target 2. Muscle 1 and 2 is held at 50 and 90% strength



(a) Without noise (b) With noise

Figure 3.25: Target 3

Target 3

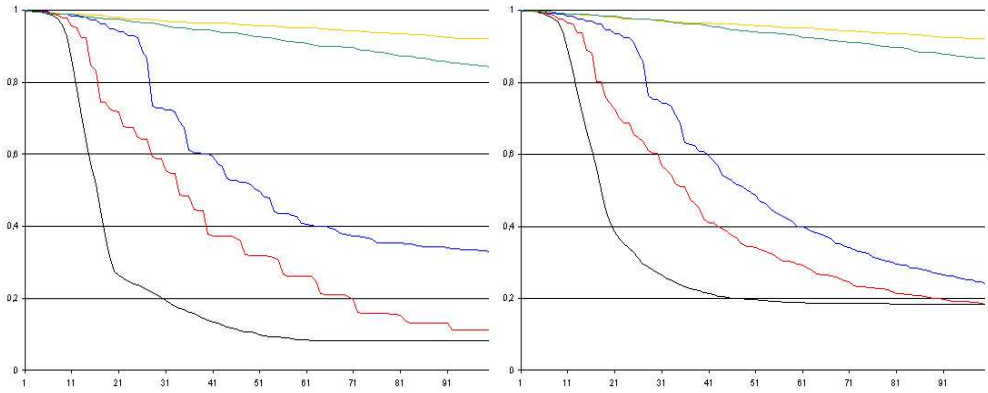
The target facial expression has muscle strengths of: 45, 45, 55, and 55%. As stated before, this is the area were the muscles are most efficient, and should therefore result in a more narrow global minimum. Figure 3.25 shows the curves with and without noise. Even thought the sum of the muscle strengths are approximately the same as for Target 2, the lowest error obtained by Stochastic Search is raised to 0.2, indicating that this target expression has got a narrower global minimum. Looking at the Push or Pull algorithm, we can see the same phenomenon as for Target 3 in facial Structure 1. One can be tempted to believe that is has been fooled by the noise to a non existing minimum, and has to small step size to search to areas with lower value. We also see that Hill Search looses in performance compared to Push or Pull and Section Search, most probably because of its slower dampening factor, which gives slower descent of the error space initially.

Target 4

Lastly we have the most difficult target. This time we have made it even worse than for facial Structure 1, as the average value of the best 3% is 0.8. This target is created by setting the muscles strength to: 25, 15, 15 and 25% Figure 3.26 shows the results achieved by our algorithms.

Again we see that the Push or Pull algorithm is the winner, though it seem to be heavily affected by the noise. The reason for this is most probably the same reason that Hill Search increases its performance, namely that noise is widening a pitfall in very much the same way as could be seen in figure 3.19 for Structure 1.

The performance of the Genetic Algorithm does not diverge so much from Stochastic Search as it did for Target 4 in Structure 1. The reason is most probably because the area with fitness higher than zero is so small. Only a small portion of the attempts to create individuals with higher fitness than zero are successful, and thus there are not enough iterations for GA to do its work.



(a) Without noise (b) With noise

Figure 3.26: Target 4

Using 500 trials

By letting the algorithms perform 500 facial updates, we can show that this is indeed the case. Figure 3.27 shows the algorithms without noise. Here we see that GA outperforms stochastic dramatically, proving our point. Further it is interesting to see that Hill Search are substantially slower in converging towards the global minimum, but nevertheless are performing a decent descent over 500 trials. This tells us that the slow reducing hyperbolic dampening factor impose a limitation on its ability to descend steep error landscapes.

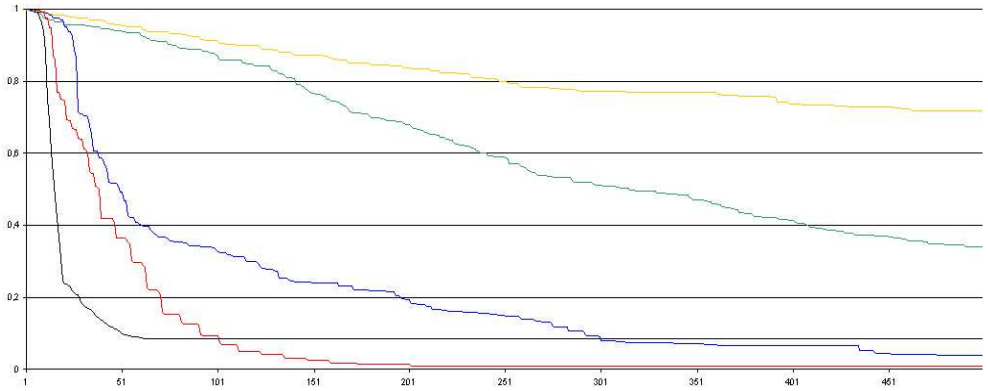
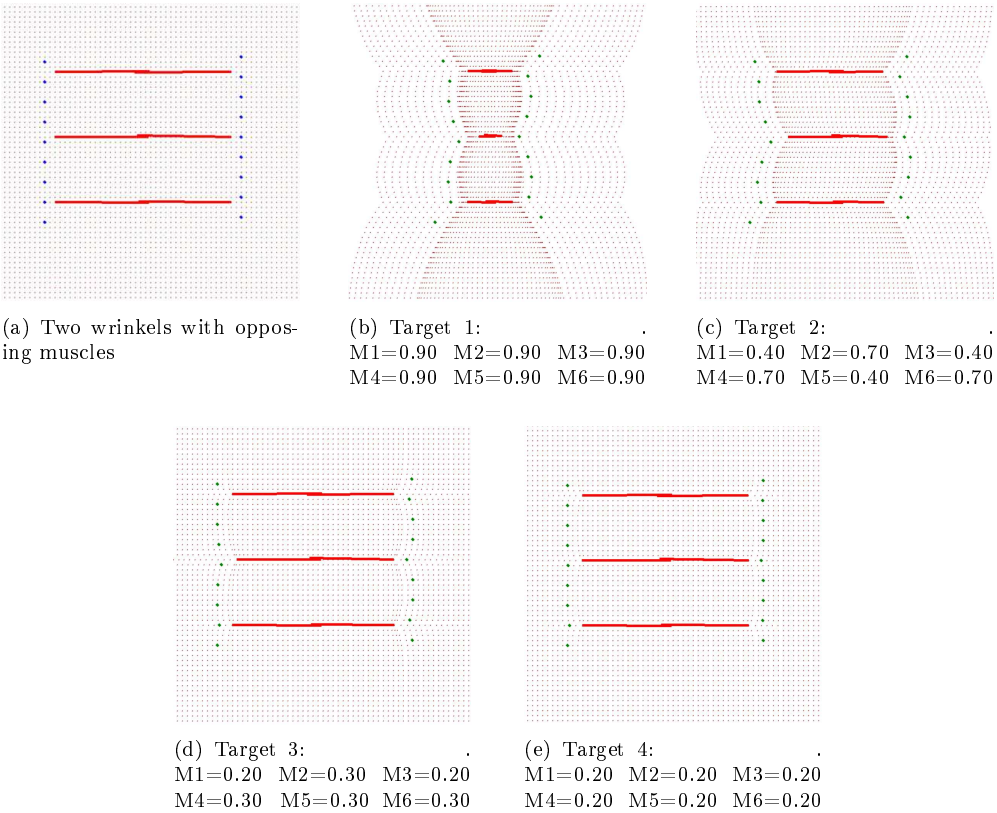


Figure 3.27: Target 4 with 500 iterations without noise

3.6.3 Structure 3 (Two wrinkles with opposing muscles)

Starting to see common patterns in the performance, the target expressions in the last facial structure seem to strengthen our theories on how the algorithms descend the error space. The facial structure consists of six muscles and 18 points, where three and three muscles are moving in the opposite direction. The structure is supposed to be a simplification of two wrinkles, pulled towards each other. In the human face this anatomical composition can bee seen between each eye brown. Figure shows the facial structure and the four different target positions.



(a) Two wrinkels with opposing muscles

(b) Target 1: .
M1=0.90 M2=0.90 M3=0.90
M4=0.90 M5=0.90 M6=0.90

(c) Target 2: .
M1=0.40 M2=0.70 M3=0.40
M4=0.70 M5=0.40 M6=0.70

(d) Target 3: .
M1=0.20 M2=0.30 M3=0.20
M4=0.30 M5=0.30 M6=0.30

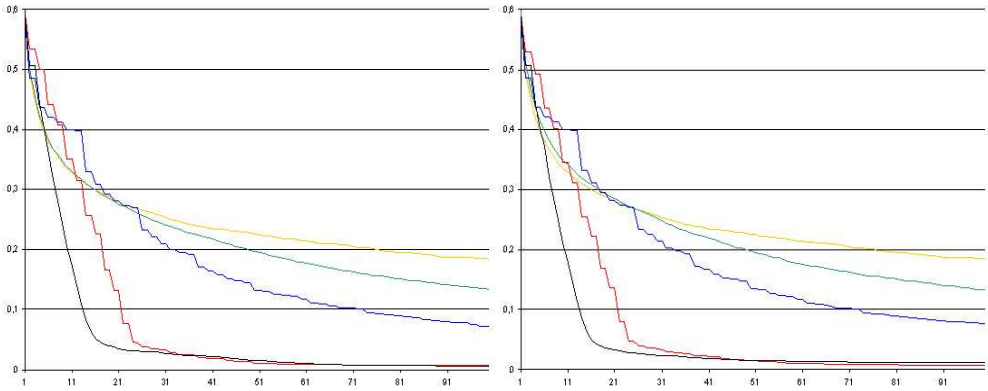
(e) Target 4: .
M1=0.20 M2=0.20 M3=0.20
M4=0.20 M5=0.20 M6=0.20

Figure 3.28: Structure 3, and its four target positions and their muscle strengths

Target 1 and 2

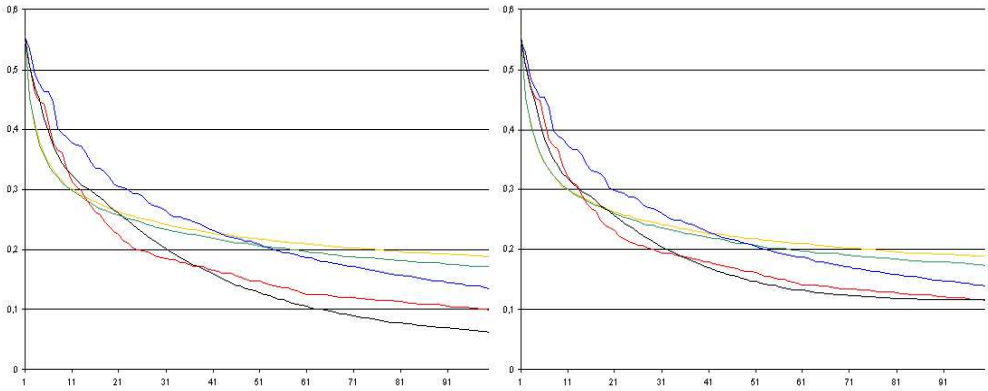
Target 1 and 2 has muscle strengths of 90,90,90,90,90,90% and 40,70,40,70,40,70% respectively. Knowing this, we should see many of the same characteristics as for the first two facial structures. Figure 3.29 and 3.30 shows the performance for each algorithm for both targets with and without noise.

The graphs all show very close to the same pattern as for the two first targets for facial Structure 1 and 2. The main difference lies in the spread in performance which has increased all the way from face 1 to face 3. This indicates that relying on feedback become more important when the error space becomes large, yielding more efficient results. While the algorithms obtained an error difference in Target 1 in Structure 1 of 0.01, the difference in performance for Target 1 in Structure 3 is 0.18.



(a) Without noise (b) With noise

Figure 3.29: Target 1



(a) Without noise (b) With noise

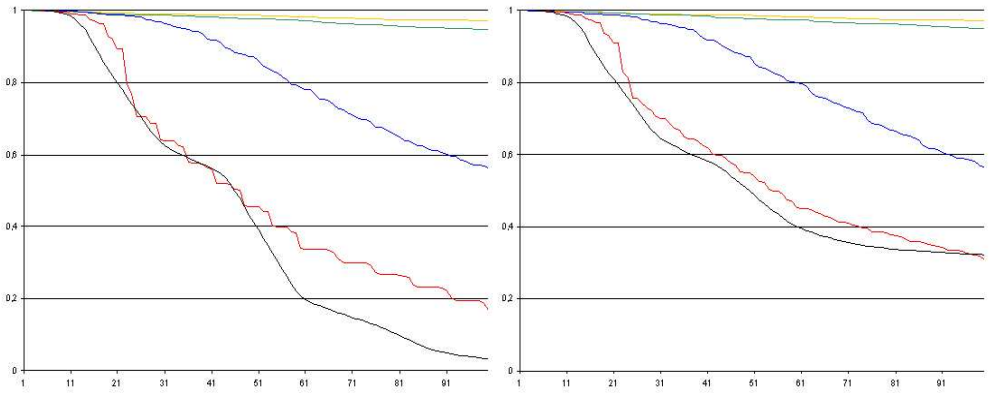
Figure 3.30: Target 2

It is also interesting to note the difference in performance between GA and Stochastic Search in these two target expressions. We see the same phenomenon yields for Target 1 and 2 in Structure 2. The reason for this divergence might be that Target 2 of both structures has got more complicated error spaces. The absence of one single region with low error value, but instead several regions of low value, might reduce the efficiency of GA. Even though we showed in figure 3.18 for Structure 1 that a region of high fitness would yield better performance for GA, several regions of higher fitness might disrupt its performance, as the algorithm uses effort in climbing regions far from the global maximum.

Noise is still no obstacle, and has almost no effect on the algorithms in these two scenarios. The only algorithm that is affected is Push or Pull in Target 2, probably due non existing minimums and small step values.

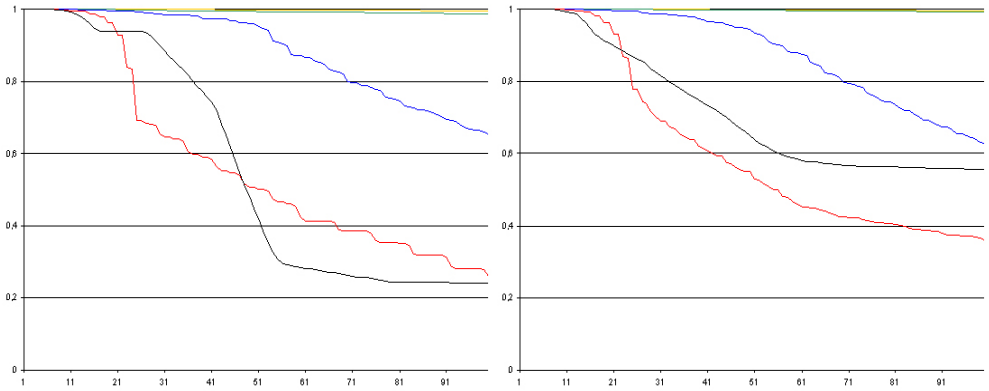
Target 3 and 4

The target muscle strength for the two last targets are: 20,30,20,30,30,30% and 20,20,20,20,20,20%. These targets are substantially more complex to solve, as can be seen from the results obtained by Stochastic Search. Figure 3.31 and 3.32 shows the development for each algorithm with and without noise.



(a) Without noise (b) With noise

Figure 3.31: Target 3

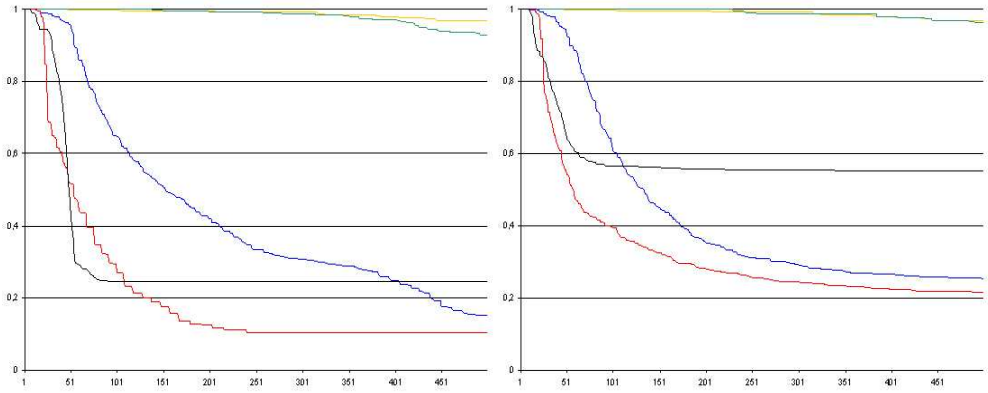


(a) Without noise (b) With noise

Figure 3.32: Target 4

Common for these two target positions is that the random based functions have almost no improvement in error at all, clearly indication that the global minimum is only small fraction of the search space. This can be easily seen by comparing their target positions with the initial position in figure 3.28.

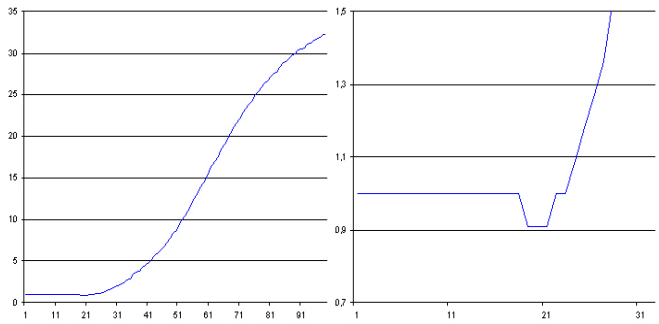
In addition we see that noise has a large effect on the Push or Pull algorithm. In Target 4 it seems like the Push or Pull algorithms stagnates above 20% error even if there is no noise present. Increasing the number of iterations clearly shows that this is indeed the case. Figure 3.33 show target 4 with 500 iterations, with and without noise.



(a) Without noise (b) With noise

Figure 3.33: Target 4 with 500 iterations

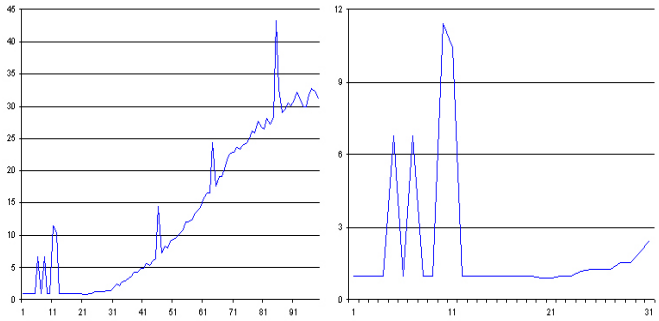
Our theory has been that Push or Pull get stuck in a plateau, but can the error landscape really be that flat. In Target 4 this is indeed the case, but it is also a result of the resolution in the simulator. Holding all muscles at zero strength, and moving muscle 3 from 0-100% strength, gives us the error function of muscle 3. This error function is shown in figure 3.34.



(a) Entire error function (b) Region below an error of 1.5

Figure 3.34: Error function for muscle 3 without noise. The error obtained is along the vertical axis, while the muscle strength is along the horizontal axis.

Seeing this error function, one quickly realizes that zero muscle strength will give a reduction of the error from almost every initial position. Once the Push or Pull algorithm has set the muscle strength to zero, it will reduce it step size in the next iteration from 0.5 to 0.25%. This value still yields a higher error, so in the next iteration the algorithm tries 12.5%. This time it jumps over but misses the ditch around 20%, getting the algorithm stuck in the flat plateau, hence the stagnation of the algorithm above 20%. The same theory also holds for the Section Search algorithm. Even though less of the muscles get stuck in the plateau, the algorithm stagnates around 10% error. Adding noise to the face, makes this plateau even worse to traverse, as it is completely soaked with irregularities as shown in figure 3.35.



(a) Entire error function

(b) The "plateau" soaked with irregularities

Figure 3.35: Error function for muscle 3 with noise. The error obtained is along the vertical axis, while the muscle strength is along the horizontal axis

Hill Search on the other hand, reduces it step with logarithmic value making it more resistant to narrow minima surrounded by flat regions. Looking at GA, we see that the area with fitness higher than zero is so small that even after 500 iterations the algorithm still doesn't perform much better than pure Stochastic Search.

3.7 Performance summary

3.7.1 Hill Search

The Hill Search algorithm was implemented in an attempt to traverse the error landscape in a more efficient manner than the other two algorithms based on feedback error. Due to the fact that finding the true gradient for in an N-dimensional error space would require $(3^N - 1)$ trials, this algorithm was implemented in very much the same way as the two other algorithms.

Performance

Looking at the results from the simulator, we see that the increased overhead in finding out which muscle to pull first did not yield an increased performance, at least not when implemented with a hyperbolic dampening factor. However, the slow decrease of the dampening factor makes the algorithm more resistant to narrow ditches and alike. This increased ability to descend more complicated error spaces comes at a price, as the algorithm in most cases converges more slowly. In the first facial structure, the step value at the 96th iteration is $1/24 = 4.1\%$. For the 6 muscle example, the step value at the same iteration has increased to $1/8 = 12.5\%$, clearly imposing problems on the achievable accuracy given 100 tries.

Conclusion

In can seem like the overhead in finding the best among several good paths before making a move, isn't worth the trouble. Using a more aggressive dampening factor might increase the algorithm's efficiency, but it might also result in less noise resistance and decreased ability to descend difficult error landscapes.

3.7.2 Section Search

Performance

The Section Search algorithm was created to give a quick descent of the error space, while at the same time making sure that the algorithm didn't get stuck in a plateau or alike. Its increased overhead in getting trapped in the error landscape leaves it outperformed by the Push or Pull algorithm in most cases, but it is in contrast to Push or Pull quite resistant to the random noise added by the simulator. Noise will only make the algorithm evaluate the same of the error space once more, removing the effect of noise after a while. However, as we saw in the last target of facial Structure 3, the algorithm can be fooled by a plateau and a surrounding ditch, if this one is narrow enough.

Conclusion

The algorithm is both quite noise resistant and quick. Normally it does not get stuck in the error landscape, and the algorithm performs best or second best over the 100 iterations period in most cases.

3.7.3 The new Push or Pull algorithm

Performance

Our Push or Pull algorithm is in many cases the fastest algorithm when it comes to descending the error space. Nevertheless, there are two main obstacles with this algorithm.

1. It has great problems in traversing flat regions
2. It is heavily affected by the noise added in the simulator.

Tweaking

The first issue can be easily solved by allowing the algorithm to move sideways, though this might give a slower descent in earlier iterations. However, noise will still be a problem in flat regions as the algorithm sooner or later will experience that noise reduces its step value to a level that is no longer productive. This problem can of course be removed by setting a lower limit on how small step value the algorithm is allowed to use. Nonetheless, large flat regions affected by noise will still present a huge obstacle for the algorithm.

Conclusion

The algorithm is the most efficient in many cases, but it comes at a price of low robustness. Noise and flat regions represent huge challenges for the algorithm.

3.7.4 Genetic Algorithm

Performance

As indicated by the results form the simulator, the Genetic Algorithm outperforms pure Stochastic Search under certain scenarios, but under normal circumstances its performance is only slightly better. Most probably, the main reason for this is the shape of the error landscapes which are quite simple. Once small regions in the error landscape yield substantially higher fitness than surrounding areas, the Genetic Algorithm increases its performance compared to Stochastic Search. As the error landscapes in the simulator only got one global minimum that often is quite accessible, GA doesn't gain that much profit by using its evolutionary techniques.

Tweaking

However, our Genetic Algorithm is not tweaked to its best, and there are several other ways to implement the algorithm that might improve its performance. Two obvious ways to increase its performance includes:

1. Changing the way we perform cross over. We have used three cross over points between the individuals, but these are not restricted to any specific parts of the chromosome. Cross over could instead be done between muscle pairs which could be used to preserve the correct genes, i.e. the correct muscle strengths amongst the fittest individuals.
2. Using Gray coding. Our Genetic Algorithm uses binary coding, and mutation of a single bit can have a large impact depending on where it is situated. If we used Gray coding instead of standard binary coding, mutation of a single bit would give values that are closer to the original binary value[38], an thus reduce the randomness introduced by mutation.

Conclusion

Evolution has had billions of years doing massive multi parallel computation before ending up with smart structures and mechanisms. Even though these mechanisms might be extremely powerful over time, they might not be the right way to go when trying to reduce an error in the least number of steps. Indeed there are many other schemes to tweak GA [8][16], but given the circumstances for which the algorithm has to work, i.e. few number of trials, we do not see the great potential in using this method.

3.8 How to increase the performance

A hybrid algorithm

Looking at the results from the simulator, one could be tempted to create an algorithm that has a rapid reduction of its step value initially, but that uses mechanics to avoid getting trapped after a specific error was obtained. A combination of Push or Pull and Section Search could for example be used, e.g. by letting Push or Pull become Section Search below a specific error. Such an algorithms could correct large imperfections quickly while using more time on delicate adjustments.

Utilizing more of the information available in the feedback can also be used to reduce the number of steps to reach the target state.

Size of error space

Performing some simple tests in the error landscape could give some additional information about the error landscape such as size and topology. Pulling the muscles at full strength could be used to measure the size of the error space, while intermediate muscle strengths could be used to divide the error space into smaller regions. This additional information could be used to adjust the initial step value, reducing the number of online trial substantially.

Gradient

Besides, there is unused information in the reduction or increase of error obtained by each online trial. This additional information could also be used to set the size of the step value. If the algorithm approached a flat region, the step size could be increased, while it could be reduced in steep regions. However, making decisions on this feedback might be dangerous, as noise can corrupt the feedback. The amount and size of noise present sets limits on how useful feedback from individual examples might be, and care should be taken when using this additional information in improving the algorithms performance.

The error measure

Besides creating other algorithms and using more of the information available in the feedback, changing the way error is defined might have an even larger impact on the algorithms performance.

To simplify the implementation and computational challenge for the simulator, the error in the simulator is calculated as the sum of all points distances. While this provides a simple error measure, the feedback information is scarce.

Using a coordinate system

An easy improvement would be to use a coordinate system with the target position as the centre position, e.g. a Cartesian coordinate system with positive and negative

values, telling the android if it has pulled the muscles too far, and not only the distance from the target position.

Weighting of points

To simplify the task for the algorithms, the points could be weighted as well, making some points more crucial than others. The most important points are the ones close to the muscles, as all other points are positioned as a function of their position. These points could therefore have a larger influence on the error calculation than surrounding points, simplifying the error space.

3.9 A different scheme for creating adaptive facial behaviour

While the task for the algorithms so far has been to reduce the number of online trails, other methods for adjusting the face might give even faster results.

A new performance measure

We have earlier stated that the time consuming part will be to generate the face expressions, and not the calculation made by the algorithms. Instead of measuring the performance in the number of facial expression tried out, the performance of an algorithm should be measured by weighting the time to reach a new face expression. In other words, how long time in seconds the face uses before converging towards a target expression.

Using continuous feedback

Instead of using the feedback from images or other sensors in a quantified manner, the feedback could be looked upon as a continuous stream, updating the skin many times per second. Using for example video feedback will thus give a totally different set of criteria to how well the algorithm performs. 10 seconds of video stream could contain several hundred different facial positions, giving new opportunities for the algorithms.

A different challenge

To reduce the time spent on updating, the algorithms need to pull the muscles in a manner that reduces the movement of the skin, and also the number of times the skin is accelerated. This can be achieved by reducing the speed of the muscles, so that feedback from the video stream gives close to correct information about the current state.

By creating algorithms that moved the contracted the muscles in a uniform direction, the time spent on moving the skin around could be reduced. In close analogy to the Push or Pull method implemented in our simulator, an algorithm that updated one muscle at a time by either contracting or relaxing the muscle until error stopped decreasing could turn out to be quite efficient. However, as we have seen in the simulator, moving one muscle to a new position affects the error of the other muscles. Depending on how the error is defined, methods could be created in order to reduce the chance of overcorrection made by each muscle.

Nonetheless, updating the facial expression using a continuous video feedback might seem more reasonable than using pictures or quantified feedback, but we have omitted the subject in this paper in order to set constraints on the topics for this thesis.

Chapter 4

Creating an adaptive face with an internal model

As we have seen, using pure algorithms in order to achieve adaptive facial behaviour demands online trial and errors, and can be quite time consuming. Better or smarter algorithms will reduce the number of trials, but they would still have to perform the search online. However, if exact continuous feedback is available from sensors in the skin, internal models [20, 18, 19] could be used to achieve adaptive facial behaviour. Internal models enables off line calculation, thus reducing or completely removing online trial and error.

4.1 Internal models

As mentioned in chapter 1, the phrase internal model is used to distinguish the actual movement performed by a muscle, and the virtual representation of the same task done in the nervous system. An internal model is capable of predicting the response of performing an action without executing the action. These predictions can therefore be used to control or guide a system in order to perform specific tasks. If a human picks up an apple to grab a bite, internal models in the brain can predict how to pull each muscle to perform this action. Internal models are classified in two categories, namely forward and inverse.

Forward models

A forward model is a model that predicts what will be the outcome of a specific motor command. It is believed that forward models exist in the human brain [17]. When the brain sends commands to its muscles, an efferent copy produced in parallel is sent to another part of the brain, predicting the sensory feedback of the ensuing action. It is believed that the central nervous system uses this prediction in several ways [17]. For the example of picking up an apple, a forward model could predict the next position of the arm, given the current position and a specific muscular input. Figure 4.1 shows a drawing of a forward model.

Inverse models

Inverse models are the opposite of forward models, and relates to how the muscle shall be pulled in order to achieve the desired action. It is the inverse models that suggest how and when to pull the muscles in order to take a bite of the apple. Inverse models therefore plays an important role in motor control. A particularly clear example of an inverse dynamic model arises in the vestibulo-ocular reflex (VOR). The VOR couples the movement of the eyes to the motion of the head, thereby allowing an organism to

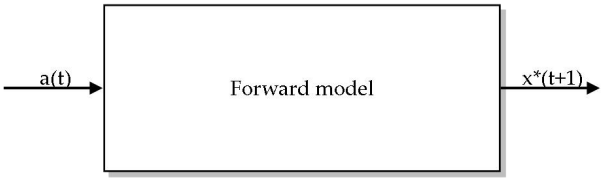


Figure 4.1: A forward model. a(t) is the action(muscular) command to the system at time t. x*(t+1) is the predicted next state of the system

keep its gaze fixed [20]. This is achieved by causing the motion of the eyes to be equal and opposite to the motion of the head.

The task of an inverse model is thus to find the correct motor command in order to achieve the desired motion or result. Figure 4.2 shows a drawing of an inverse model.

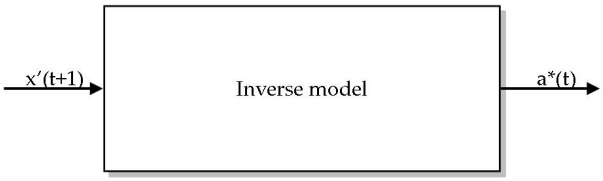


Figure 4.2: An inverse model. x'(t+1) is the desired next state, and a*(t) is the predicted muscular command needed to move the system to state x'(t+1)

4.2 Using internal models to achieve adaptive facial behaviour

In the case of an android head, we see that a forward model could be used to predict how the skin would react to a certain muscle contraction, i.e. the new positions of structures in the skin given a specific muscle configuration. The inverse model on the other hand could predict how to pull the muscles in order to move the skin to a target position, i.e. come up with the desired facial expression.

The task of creating adaptive facial behaviour, can be considered a control problem, namely how to pull the muscles in order to achieve the desired face expression. This is exactly what can be performed by an inverse model, and it would therefore be more tempting to implement an inverse model than a forward model to achieve adaptive facial behaviour.

What can be solved

It is important to realize that if an inverse model is to be used to achieve adaptive facial behaviour, it must be able to learn the inherent dynamics of the face during different conditions. Therefore the feedback should to be given through sensors in the skin. If the android relies on visual feedback, there is little performance gain in using internal models compared to optimization algorithms. In theory, an internal model relying on sporadic visual feedback could be used to predict the results of fatigue and create new facial expressions without trial and error. However, adjusting for tear and wear, and correcting errors such as temporary noise would require visual feedback, as the model consequently is out of date.

Given continuous and exact feedback from sensors in the skin, internal models could in theory cope with all the problems of having only pre defined face expressions mentioned in section 1.1. The question becomes how to create such a model, and how to make it work.

4.3 Creating an inverse model with Direct Inverse Learning

There are several ways to construct and implement an inverse internal model. One simple way is called Direct Inverse Learning (DIL) [20, 19]. DIL works by observing a set of input output pairs of a system, and use the observed results to train itself to perform the opposite action.

How to implement Direct Inverse Learning

In the case of an android face, the system could be the muscles and the skin. The input to this system could be the muscular command to move the skin, and the output could be the resulting skin position. By observing the face during normal operation, a list of input output pairs could be created. By reversing these input output pairs, the model could train itself to perform the opposite action. The inverse model would therefore use the resulting skin position as input, and the muscle strength causing this position as output. Given a large and representative training set, the inverse model should after a successful training be able to predict the muscle force needed to reach skin positions not present in the training set. Figure 4.3 shows the principles behind Direct Inverse Learning.

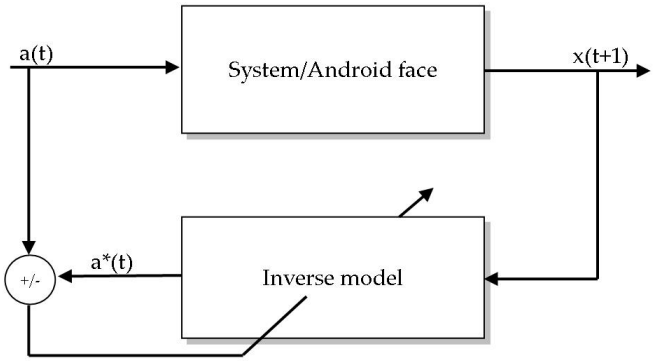


Figure 4.3: Direct inverse learning. a(t) is the muscular command to the system, while x(t+1) is the next state of the system. a*(t) is the predicted muscle strength moving the system to x(t+1). The discrepancy between a(t) and a*(t) is used to train the inverse model

How to create the learning system

With a method for creating an internal model, we need a mechanism for learning the relationship between the input output pairs. Artificial neural networks (ANNs) are learning systems inspired from biology that can be trained to map correlations between data, and as such can be used to model, predict or classify data in many different areas [31]. The field of ANNs is huge, and there are many different kinds of network structures and training algorithms available. We have chosen to use a two layered feed forward network trained by Back Propagation as a model for the Direct

Inverse Learning. The next section 4.4 introduces the theory behind ANNs and Back Propagation. For readers familiar with the subject, the next section can be skipped without loosing any form of consistency.

4.4 Artificial neural networks

Artificial Neural Networks [14, 31, 25, 2] are inspired by the biology of the human nervous system, and originated in an attempt to copy the human brains information-processing capabilities. To understand the composition and theory behind ANNs, a brief introduction to the workings of biology is desirable.

Our nerves system consists of a collection of neuron units communicating with each other through dendrites and axons. Figure 4.4 show a simplified model of a neuron. The neurons consists of three parts, a neuron cell body, branching extensions called dendrites for receiving input, and an axon that carries the neuron's output to the dendrites of other neurons. The area where two neurons interconnect is called a synapse. A neuron receives signals from nearby neurons through the synapses in its dendrites, and sends out signals through the synapses in its axons. These synapses can be either inhibitory or excitatory and their ability to influence other neurons may vary form synapse to synapse. That is, a neuron may consider an excitatory signal from one neuron twice as important as an inhibitory signal from another neuron due to the different properties of the synapses. If the excitatory influences from these dendrite synapses are sufficiently dominant, then the neuron fires and sends action potentials through its axon. Action potentials are spikes of current, enabling the synapses of transmitting signals form a neuron to the next. These action potentials therefore activate or inhibit other neurons via their respective outgoing synapses.

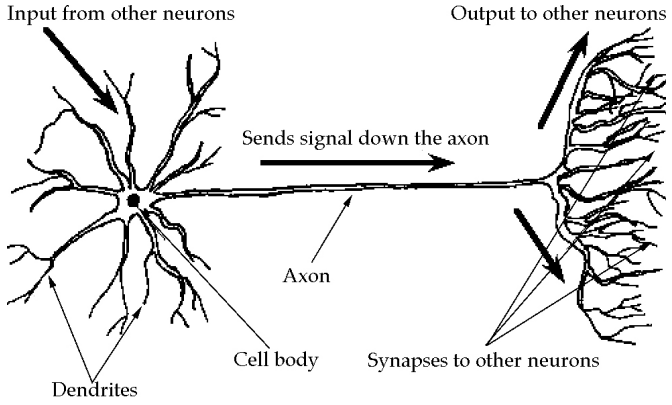


Figure 4.4: A simple model of the neuron

4.4.1 Basics of application driven neural networks

Based on some of this knowledge, McCulloch and Pitts proposed the first model of neural nets in 1943[14]. Figure 4.5 shows a simple model of their artificial neuron using a threshold activation function. i_n is the inputs, w_n is the weight of the respective input, and o is the output. The inputs of the artificial neuron are analogous to signals received from other neurons, while the weights are analogous to the importance of each such signal due to different properties of the synapses. The artificial neuron summarises the weighted sum of all its inputs, and outputs 1 if the result is greater than some threshold and -1 otherwise. This can be related to a neuron that fires an action potential if the output is 1, while and -1 means that the neuron does nothing.

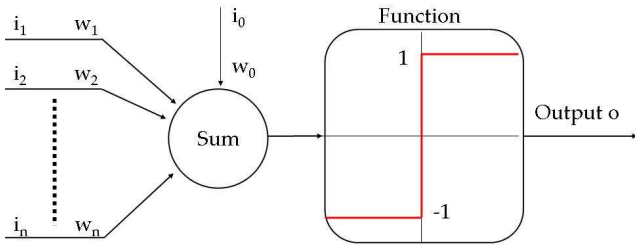


Figure 4.5: McCulloch and Pitts neuron. i_n is the inputs, w_n is the weight of the respective input, and o is the output

In order for a network to perform a classification, prediction or analysis of data, it has to be trained. Training of a neural network is accomplished by adjusting each neurons weights, so that the network outputs the desired value for any given input. In order to train the network we have to have training examples, that is, problems or coherence in data we know the answer to. We could train a network to understand the function $f(x) = x^2$ by giving the network these examples: $2^2 = 4$, $3^2 = 9$, and $4^2 = 16$. A correctly trained network, with the right network structure and weights, would output 25 if we gave 5 as input. To see how this is possible, let us take a closer look on how to make a network represent a function.

Representing a function

An easy example would be to say that the neuron in figure 4.5 has two inputs, and that we want to make it represent an AND function. That is, we want the neuron to output 1 if both inputs are 1, and -1 otherwise. Notice the extra input i_0 with weight w_0. This is the threshold value that the weighted inputs must surpass in order for the neuron to output 1. Normally we set the threshold input $i_0 = 1$, and adjust the threshold weight w_0 according to the desired threshold. The output o can then be calculated by:

Output o is 1 if: $w_0 i_0 + w_1 i_1 + w_2 i_2 > 0$
Output o is -1 if $w_0 i_0 + w_1 i_1 + w_2 i_2 < 0$

Making the network represent an AND function is easily accomplished by for example setting the threshold weight w_0 to -1, and the two input weights to: $w_1 = 0.6, w_2 = 0.6$. Finding the right weight for a single neuron is pretty easy, but if we are to construct a larger network with several neurons, we need a rule for updating the weights.

Training a network

Most neural networks have some sort of training rule whereby the weights of the connections are adjusted on the basis of presented training patterns. This training enables artificial neural networks to learn from examples, and they can therefore be used to perform a wide range of tasks, such as classification, estimation, simulation and prediction. The Perceptron Training Rule is a way of finding the weights in a one layered network with threshold activation function. The training rule forms the basis for more advanced networks and is as follows:

1. Give each weight an initial random value.
2. Iteratively apply a training example to the network.
3. Modify the networks weights whenever it misclassifies an example.

This process is repeated until the network correctly classifies all training examples. The network misclassifies an example if the output of the network is not the same as the desired output. For the neuron representing the AND function, a misclassification could be to output minus one if both inputs were one. In that case, each weight should be updated according to the rule: $w_x = w_x + \Delta w_x$, where: $\Delta w_x = l \cdot (t - o) \cdot i_x$. Here t is the target output(1), and o is the output generated by the network(-1). l is a positive constant called the learning rate in the range between 0-1. Basically what this rule says is: The magnitude of a weight adjustment should be proportional to the size of the error$(t - o)$ and the strength of the input. That is, the higher the error, or stronger the input, the more one need to adjust the weight. Finally, multiply this number with a learning rate, in order to avoid oscillations of the weights. To see that this rule works, let use an example. First we give the weights an initial random value. Let this be $w_1 = -0.9, w_2 = 0.2, w_n = -0.6$. Then we apply the input value $i_1 = 1, i_2 = 1$. The output of the network becomes:

$$o = sgn(\vec{i} \cdot \vec{w}) = sgn([1, 1, 1] \cdot [-0.9, 0.2, -0.6]) = -1 \tag{4.1}$$

The output should have been 1, not -1. By using the Perceptron Training Rule, and a learning rate of 0.1, the weight update becomes:

$$w_1 =\leftarrow -0.9 + 0.1(1 - (-1))1 = -0.7$$
$$w_2 =\leftarrow -0.2 + 0.1(1 - (-1))1 = 0.4$$
$$w_n =\leftarrow -0.6 + 0.1(1 - (-1))1 = 0.4$$

With the new weights, we see that the neuron represents an AND function.

Even though this simplified example gave an impression of how to train a neural network, a one layered network has got limited representational capacities. I order to increase the representational power of a neural network, we have to introduce hidden layers. Although the greater power of multi-layered networks was realized long ago, it was only recently shown how to make them learn. In 1985 Rumelhart, Hinton and Williams, published a learning algorithm called Back Propagation that where able to train multi-layered network with non-linear activation functions[14]. To understand Back Propagation in multilayered networks, its useful to take a look at a learning algorithm called Stochastic Gradient Descent.

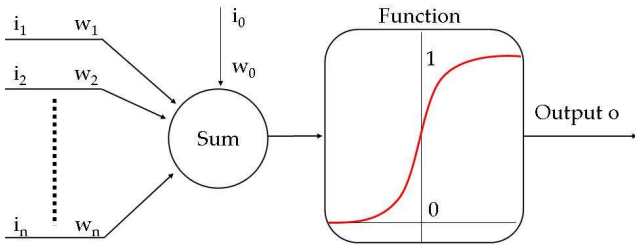


Figure 4.6: Neuron with sigmoid activation function

4.4.2 Stochastic Gradient Descent

Stochastic Gradient Descent [31, 28, 42] is important because it provides the basis for the Back Propagation algorithm which learns networks with many layers. Stochastic Gradient Descent can only be used on differentiable activation functions. Therefore, it cannot be used on networks with threshold sign functions as we have looked at so far. One could of course use networks with linear activation functions. These functions are differentiable, but such networks can only represent linear functions[14]. Introducing hidden layers doesn't help. To increase the representational power of the network,

we have to use non-linear activation functions. The most common of the non-linear activation functions is the sigmoid function. Figure 4.6 shows a neuron with a sigmoid activation function. It resembles the threshold sign function introduced by McCulloch and Pitts, but it can be differentiated. More precisely the sigmoid unit calculates the output as:

$$o = \sigma(\vec{w} \cdot \vec{i}) = \sigma(S)$$

Where:

$$\sigma(S) = \frac{1}{(1 + e^{-S})}$$

Note its output ranges between 0 and 1, increasing monotonically with its input. The sigmoid function has the advantage that its derivative is easily expressed in terms of its output.

$$\sigma'(S) = \sigma(S) \cdot (1 - \sigma(S)) \tag{4.2}$$

Stochastic Gradient Descent works the same way as the Perceptron Training Rule, but the weight update rule become dissimilar due to the fact that the sigmoid function has different properties from the threshold function. While we just stated that the weight update was $\Delta w_x = l \cdot (t - o) \cdot i_x$ for the Perceptron Training Rule, we derive the weight update rule in Stochastic Gradient Descent by analysing the error space.

The error measure

To analyse the error space, we have to define an error measure for the network. There are many ways to do this, and for the Perceptron Training Rule we said that this was (t-o). While this is a simple error measure, it introduces problems because it allows the error to obtain negative values. An error measure that has turned out to be especially convenient is:

$$E(\vec{w}) = \frac{1}{2}(t - o)^2 \tag{4.3}$$

$E(\overrightarrow{w})$ is the network error for one training example given the weight vector $\vec{w}$. Here t is the target output, and o is the output of the neuron.

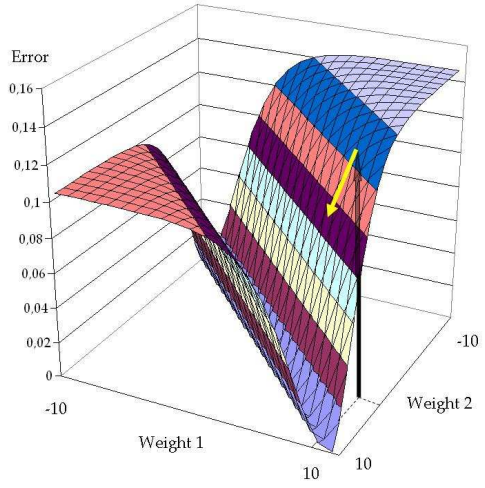


Figure 4.7: Error space for a two input neuron. the target value of the network is 0.46, and both inputs are 0.5. The weights range from -10 to 10.

Visualizing the error space

Suppose we had a network consisting of a two-input neuron with a sigmoid activation function, and applied a certain input to the neuron. By changing one weight at a time, while holding the input values constant, we could read out all possible outputs from the neuron. If we knew the network target output value, we could calculate the error for each weight configuration. This error could then be visualized in a three dimensional error space. Figure 4.7 shows a typical error space for a two input neuron with sigmoid activation function.

Deriving the weight update

Suppose a weight vector for one training example gave us the error E. The best way to remove this error would be to adjust the weights so that the error moved closer to the bottom of the valley. In other words, we want to find a Δw we could add to each weight , so that the error became zero. Stochastic gradient decent uses the partial derivative of the error function to find the vector pointing towards the bottom of the valley. The partial derivative gives the quotient of the error space for each axis, and can therefore be used to find the vector that point in the direction of the steepest gradient. Negating this vector gives us a vector pointing with the steepest descent towards the bottom of the valley. The yellow arrow in figure 4.7 shows this vector. Realize that each component of this vector is the Δw we are looking for, giving us an update rule for each weight, reducing the error as quick as possible.

We find Δw for each weight by taking the partial derivative of the error function with respect to each weight, and negating this value.

$$\Delta w_x = -\frac{\partial E}{\partial w_x} \tag{4.4}$$

The error space E can be seen as a function of the sum of the weighted inputs. For one input pattern, the sum of the weighted inputs can be seen as a function of the weights as the inputs are constant. We can therefore use the chain rule to write the partial derivative as:

$$\frac{\partial E}{\partial w_x} = \frac{\partial E}{\partial S} \cdot \frac{\partial S}{\partial w_x} \tag{4.5}$$

Where S is the sum of the weighted inputs $S = \sum_{x=0}^{n} w_x i_x$. The derivative of the sum S with respect to each weight is $\frac{\partial S}{\partial w_x} = i_x$. Inserting this into equation 4.5, we get:

$$\frac{\partial E}{\partial w_x} = \frac{\partial E}{\partial S} \cdot i_x \tag{4.6}$$

The error of the network with respect to the weighted sum of the inputs is $e = \frac{\partial E}{\partial S}$ called the "error term". This expression is widely used in the Back Propagation algorithm, and its definition is worth noting here. Looking closer at this equation, we see that the network error also can be looked upon as a function of the output. The output is also a function of the sum of the weighted inputs, and we can therefore use the chain rule once more to write:

$$e = \frac{\partial E}{\partial S} = \frac{\partial E}{\partial o} \cdot \frac{\partial o}{\partial S} \tag{4.7}$$

Now $\frac{\partial o}{\partial S}$ is the derivative of the output with respect to the sum. For the sigmoid function, this can easily be expressed in terms of its output.

$$\frac{\partial o}{\partial S} = \frac{\partial \sigma(S)}{\partial o S} = \sigma(S) \cdot (1 - \sigma(S)) = o(1 - o) \tag{4.8}$$

Inserting equation 4.7 into equation 4.8 we get $e = \frac{\partial E}{\partial o} \cdot o(1 - o)$. Now we only need to differentiate the error with respect to the output:

$$\frac{\partial E}{\partial o} = \frac{\partial(\frac{1}{2}(t-o)^2)}{\partial o} = \frac{1}{2} \cdot 2 \cdot (t-o) \cdot \frac{\partial(t-o)}{\partial o} = (t-o)(-1) = -(t-o) \tag{4.9}$$

Inserting this into equation 4.7 we get:

$$e = \frac{\partial E}{\partial o} \cdot o(1-o) = -(t-o) \cdot o(1-o) \tag{4.10}$$

This is the "error term" of a neuron, a value that becomes essential in the Back Propagation algorithm. To get the partial derivative of the error function, we have insert equation 4.10 into equation 4.6. The partial derivative for each weight in the neuron with a sigmoid activation function therefore becomes:

$$\frac{\partial E}{\partial w_x} = \frac{\partial E}{\partial S} \cdot i_x = e \cdot i_x = -(t-o) \cdot o(1-o) \cdot i_x \tag{4.11}$$

Remember that calculating the partial derivative for all the weights in the error space gives us the components of the vector which points in the direction of the steepest gradient. We want this vector to point in the direction of the steepest descent. We therefore have to negate it. The weight update rule becomes:

$$\Delta w_x = -\frac{\partial E}{\partial w} = (t-o) \cdot o(1-o) \cdot i_x \tag{4.12}$$

We have now derived a way to update the weights if the network misclassified a training example, but this still only holds for one specific input pattern.

Several training examples

Remember that the error landscape in figure 4.7 is for one neuron with fixed inputs. Different inputs changes the steepness and direction of the valley, and thus give a different error landscape. A weight combination that gives zero error for one training example might give us substantial error given a different input pattern. This is easy to see if we add several error landscapes on top of each other. Figure 4.8 show an error space made of three training examples.

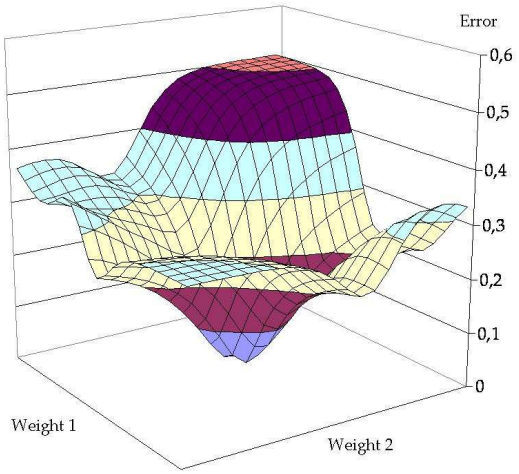


Figure 4.8: Error landscape for a two input neuron with three training examples

The learning rate

To avoid moving the weights too much when presented with at training example, we multiply equation 4.12 with a small constant called learning rate. This learning rate ensures that the weights are only adjusted a little bit towards the bottom of the valley for each input pattern. The neuron will therefore be trained to output less error for each training example, while not disrupting too much for the other input patterns. Our weight update rule therefore becomes:

$$\Delta w_x = -l \cdot \frac{\partial E}{\partial w} = l \cdot (t - o) \cdot o(1 - o) \cdot i_x \tag{4.13}$$

This is the same weight update rule as for the Perceptron Training Rule, but with an additional constant $o(1 - o)$. In fact, the weight update rule would have been exactly the same as the Perceptron Training Rule if we used a linear activation function in the neuron. The additional constant $o(1 - o)$ comes from the derivation of sigmoid function itself, as can be seen from equation 4.2.

Summary

A neuron presented to several training examples will be fully trained if the weights are adjusted in a way that makes the error become the global minimum. Stochastic Gradient Descent tries to find the global minimum by descending each training example towards the minimum error. For each training example, it updates every weight by adjusting the weight vector by a small amount in the direction of the least error for that training example. This is accomplished by partial differentiation of the error space. While this is only possible to visualise for a two input neuron, it holds for any number of inputs. For a neuron with three inputs, the error space would be four dimensional and so fourth.

Networks with sigmoid activation functions may have several local minima. The gradient descent algorithm is therefore not guaranteed to find a weight vector that gives the minimum error. It is only guaranteed to find a local minimum.

For simplicity, we have so far only looked at networks with one neuron. Both Stochastic Gradient Descent and the Perceptron Training Rule works for one layered networks with any number of neurons. This is because each unit only adjusts the weights reaching it selves. The weight adjustment on one unit has therefore no influence on the output of other neurons. The only difference becomes that you have to calculate the error of several outputs. The mathematical notation becomes somewhat more complicated however.

4.4.3 Back Propagation

The representational power of one layered networks are limited. If we are to use neural networks for more advanced problems, we have to introduce hidden layers. Multi-layered networks with one hidden layer can represent any continuous function, while networks with two or more hidden layers are capable of representing any function with arbitrary accuracy [31]. Figure 4.9 shows a multilayered network with 8 inputs, and 3 outputs. Its a two layered network, consisting of one hidden layer.

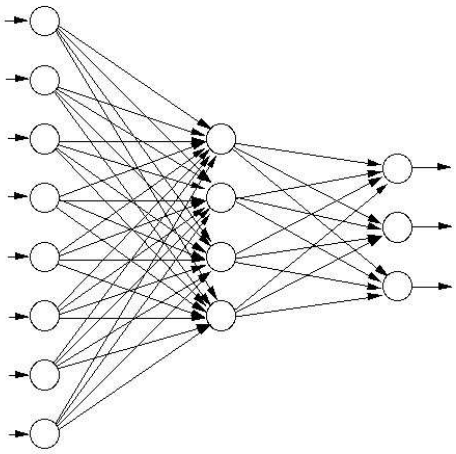


Figure 4.9: Two layered network with 8 inputs and three outputs

Back Propagation

Back Propagation [14, 31, 25, 2] is the most common way of learning multi-layered feed forward networks, and is based on Stochastic Gradient Descent. As with Stochastic Gradient Descent, it uses partial derivation of each neurons error space to find the direction of the steepest gradient. By negating and multiplying this vector by a little amount, Back Propagation descends the error space exactly the same way as gradient descent, but introduces a weight update rule for the hidden units as well.

When we take a look at a multi layered networks, we see that the output units can be looked upon as a one layered network. Whether there are nodes or inputs before the last layer is irrelevant for the output layer. As long as the values are constant, they can be treated the same way. The weight update rule for these units should therefore be exactly the same as for the neurons with sigmoid activation functions trained by Stochastic Gradient Descent. The question becomes how to derive an update rule for the hidden units, as we have no clue about their target value. We only know the target value of the entire network, and not the hidden nodes.

How to propagate the error

By looking at the network structure, one realizes that the error from a hidden unit only affects neurons downstream form that unit. That is, neurons that has the output of the hidden neuron as input. Figure 4.10 shows a two layered network with one input, three hidden nodes and three output nodes. As we can see, the error generated by the hidden node in red only affects is own output, and the output of the output neurons. It is obvious that the size of each weights matter when it comes to how this error propagates through the network. By analyzing the error space of the hidden units, and the amount of error and the size of the weights, we can derive a measure for the error created by a hidden neuron.

According to the Back Propagation algorithm, the weight update rule for the hidden units should be:

$$\Delta w_{xy} = -l \cdot (\sum_{z \in Downstream\ y} e_z \cdot w_{yz}) \cdot o_y(1 - o_y) \cdot i_{xy} \tag{4.14}$$

Here w_{xy} means the weight from node x to node y. o_y is the output of node y, and i_{xy} is the output of node x. e_z is the "error term" of node z which is every node downstream of node y. By downstream we mean all nodes that are directly coupled to node y, and that have node y's output as input. Note that the "error term" is

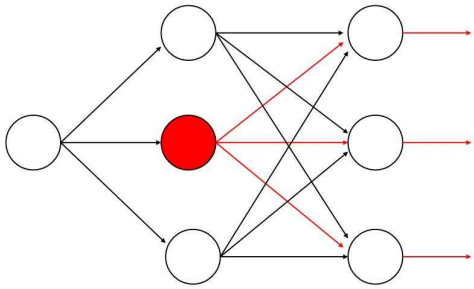


Figure 4.10: One input network with three hidden nodes and three outputs. The error from the hidden node in red is only propagated to the output nodes

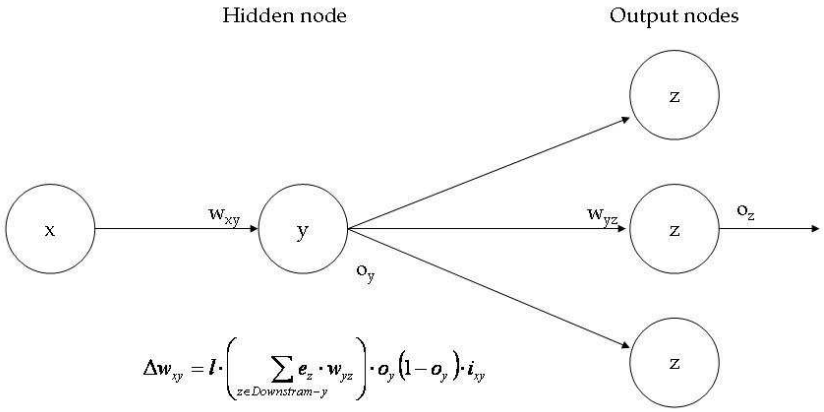


Figure 4.11: The weight update for hidden units

not the error generated by the node, but the expression we derived when we looked at gradient descent. The error term is defined as the derivative of the network error with regard to the nodes weighted sum of inputs $e_y = \frac{\partial E}{\partial S_y}$, and is just a convenient measure to simplify the equation. Figure 4.10 shows the notation of the weight update rule in graphical form. In other words, Back Propagation states that; **"The "error term" in a hidden unit is the weighted sum of the "error term"s of all units downstream of y, multiplied with the derivative of its own activation function**." By multiplying this error term with the learning rate and the input to the neuron, we get the weight update rule for the hidden units.

The error term of a hidden node

In order to derive the error term of a hidden node, we have to know the error term of the neurons downstream of that node. Recall that the total error of the network can be calculated, as we know the target output of the network during training. As we have already stated, the output nodes can be looked upon as neurons in a one layered network. We can therefore use the definition of the error term from Stochastic Gradient Descent for these neurons. For an output unit z the error term becomes:

$$e_z = -(t_z - o) \cdot (1 - o_z) \tag{4.15}$$

The error term of all output units can then be back propagated through the network, one layer at a time, hence the name Back Propagation. A network can therefore have as many layers as desired, since we just calculate the error term for all nodes one layer

at a time, starting with the output layer. It is worth noting that the weight update rule for the hidden nodes only summarizes the error terms of the neurons directly connected to the hidden node, and not all neurons in following layers. This is because the errors terms in all these layers have already been calculated. To summarize the error term in all successive nodes would be an overestimation. Figure 4.12 summarises the definition of the error term and the weight update rule for a hidden and an output node in a network learned by the Back Propagation algorithm.

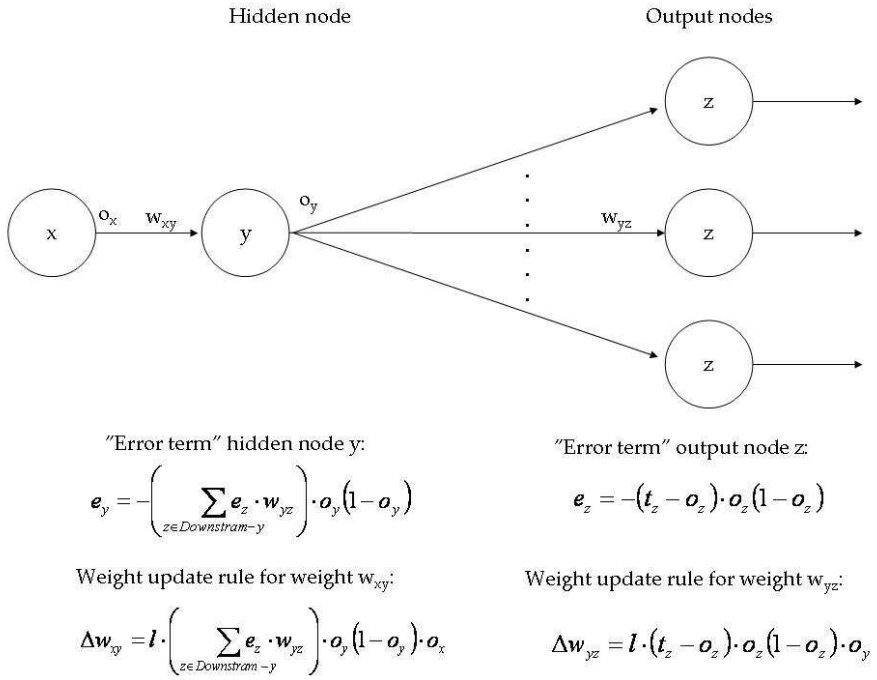


Figure 4.12: . The error term and weight update rule for hidden nodes and output units

Deriving the weight update rule for the hidden units

We can derive the update rule for the hidden units the same way as for neurons in Stochastic Gradient Descent. The error space for a two input neuron was easily visualized, but to visualize the error space for a hidden unit become more complicated. Nevertheless, the logic still holds, and follows the same pattern. We still want to find a vector pointing towards the steepest descent. Remember that we could find the components of the vector pointing in the direction of the steepest gradient by taking the partial derivative of the error space. This vector is Δw, and each component are of the form Δw_{xy}, where w_{xy} means that this is the weight from node x to node y. As for gradient descent the Δw_{xy} can be derived from partial derivation of the error space.

$$\Delta w_{xy} = -\frac{\partial E}{\partial w_{xy}} \tag{4.16}$$

Remember that the error space E can be seen as a function of the sum of the weighted inputs to the neuron. We can therefore invoke the chain rule to write the partial derivative as:

$$\frac{\partial E}{\partial w_{xy}} = \frac{\partial E}{\partial S_y} \cdot \frac{\partial S_y}{\partial w_{xy}} \tag{4.17}$$

The expression $\frac{\partial E}{\partial S_y}$ is the error term of node y. The term $\frac{\partial S_y}{\partial w_{xy}}$ can still be written as i_{xy}, hence:

$$\frac{\partial E}{\partial w_{xy}} = e_y \cdot i_{xy} \tag{4.18}$$

We now have to find the value of the error term. We do not readily know the contribution of S_y on the output error E of the network. On the other hand we realize that S_y can only influence the network error through the units downstream of it self. We can therefore invoke the chain rule to write:

$$e_y = \frac{\partial E}{\partial S_y} = \sum_{z \in Downstream\ y} \frac{\partial E}{\partial S_z} \frac{\partial S_z}{\partial S_y} \tag{4.19}$$

The chain rule made it possible to derive the factor $\frac{\partial E}{\partial S_z}$ which is the error term for each unit downstream of y. The second factor $\frac{\partial S_z}{\partial S_y}$ isn't directly solvable, but we see that the weighted input sum of each node z can be looked upon as a function of the output of node y. By inserting the error term for node z, and using the chain rule once more, we can write the equation as:

$$e_y = \sum_{z \in Downstream\ y} e_z \frac{\partial S_z}{\partial S_y} = \sum_{z \in Downstream\ y} e_z \frac{\partial S_z}{\partial o_y} \frac{\partial o_y}{\partial S_y} \tag{4.20}$$

Looking closer at the new differential, we see that the first factor is easily solved; it is just the weight on the incoming connection from node y to node z:

$$\frac{\partial S_z}{\partial o_y} = w_{yz} \tag{4.21}$$

The second term of the differential is the derivative of the sigmoid activation function. This differential can therefore easily be solved:

$$\frac{\partial o_y}{\partial S_y} = o_y(1 - o_y) \tag{4.22}$$

Inserting equation 4.21 and 4.22 into equation 4.20, gives us:

$$e_y = (\sum_{z \in Downstream\ y} e_z \cdot w_{yz}) \cdot o_y(1 - o) \tag{4.23}$$

We have now derived the error term for a hidden unit by analysing the influence from the neuron's weight on the error space of the network. Inserting this into equation 4.18 we get the partial derivate of the error space with regards to that weight.

$$\frac{\partial E}{\partial w_{xy}} = e_y \cdot i_{xy} = (\sum_{z \in Downstream\ y} e_z \cdot w_{yz}) \cdot o_y(1 - o) \cdot i_{xy}$$

Remember that Back Propagation descends the error space the same way as Stochastic Gradient Descent. We therefore have to negate this vector, and multiply it with the learning rate in order to get the weight update rule for a hidden unit:

$$\Delta w_{xy} = -l \cdot (\sum_{z \in Downstream\ y} e_z \cdot w_{yz}) \cdot o_y(1 - o) \cdot i_{xy} \tag{4.24}$$

Summary

As we see, both Back Propagation and Stochastic Gradient Descent uses the same logic when it comes to deriving a weight update rule. They both descend the network error space towards the least error by a little amount for each training example. Multi layered networks can have many local minima, and Back Propagation is not guaranteed to find the global minimum. However, Back Propagation have shown to be highly successful in many areas, and the discovery of the algorithm gave the field of artificial neural networks a new boom in the mid 80s. Since then, ANNs have been used widely in commercial products, both in software and hardware [28, 31].

4.5 The experiment

To see the effect of implementing Direct Inverse Learning with a neural network as a model on the simulator, we use the same facial structures as the ones searched by the algorithms in section 3.6.

Each facial structure has its own internal model, as their "physics" are all different. *The challenge for each inverse model is to predict the muscle strength needed to move each structure from its initial position (when all muscles are relaxed) to their respective target positions.* The models are trained by observing the facial structures during normal operation. This is achieved by letting the simulator generate random muscle strengths, and read out the resulting position of each structure. After training, the models get one attempt to reach each of the four target expressions.

4.5.1 Implementation of the neural network

To enable Direct Inverse Learning to map a relationship of the physics in the simulator, each facial structure has its own neural network. Figure 4.13 show a simplified drawing of how the neural network using Direct Inverse Learning is implemented in our simulator.

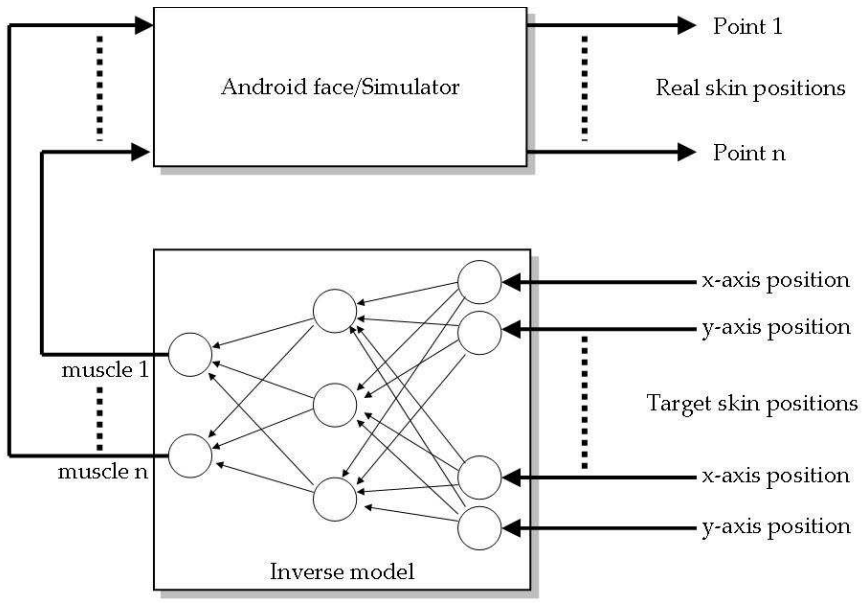


Figure 4.13: Implementation of the neural network as a inverse model using Direct Inverse Learning. The inputs to the network is the X and Y axis positions of each points, while the output of the network is the predicted muscle strength.

The network structure

As the number of points and muscles are not the same in the three different facial structures, the network becomes different for each structure. However, the methodology for creating the networks is the same. Each network is created by using two input nodes for each point in skin. Each input node refers to the X or Y coordinate of the point in the simulator. The network uses one hidden layer, and there is one output node for each muscle. A face structure with ten points and three muscles will therefore have twenty inputs and three outputs.

Network size

To obtain good generalization ability, we wanted to keep the number of hidden nodes to a minimum. Increasing the number of hidden nodes can reduce the error on the training set, but it might also reduce the networks ability to generalize the target function [14]. We found the minimum number of hidden nodes to be 40 for structure 1, and 20 for structure 2 and 3. Using more hidden nodes had little effect on the performance, but increased training time substantially.

Training the network

To create a training set for the internal model, we let the simulator pull the muscles by random. Each network was trained by creating 50 random face expressions. By reading the strength of each muscle, and the resulting positions of each point in the skin, we created an array with the input output pairs for each structure. The values in this array was inversed to train the neural network, and the weights were adjusted using Back Propagation. The maximum number of Back Propagation iteration was set to 10.000. To avoid overtraining, a separate validation set of 50 facial expressions was used to benchmark the performance of the network.

4.5.2 How to interpret the results

Even though the inverse model is exposed to the same challenges as the algorithms, there is little information in comparing their performance. Their inner workings are totally different, and to compare their performance would be the same as comparing the speed of a boat and a car. It will depend on whether the trials are done on solid ground or not. Optimization algorithms will find the goal state sooner or later, but may require unlimited number of trials. The neural network on the other hand is not guaranteed to find a solution, but needs only one trial.

To see the effect of using an inverse model, the error obtained trying to reach each target, are instead compared with a random pull.

Number of iterations

For each target expression, the process of training and using the network to suggest muscle strengths is performed 100 times, and not 1000 as for the algorithms. The time consuming nature of training a network have made us set these restrictions, but ultimately we would have trained the network 1000 times. Each time the network is trained, it receives different training examples with different initial weights. Each of these networks gets to predict the muscle force needed to reach each of the four different target facial expressions. The results of these muscle contractions are averaged over the 100 runs. This entire process is repeated three times, giving an idea about the deviation of the results.

Error calculation

As for the error calculation in chapter 3, we assume that the maximum obtainable error is 1, as this is the error from the initial position of each structure. However, we have also made an additional error calculation where the maximum error is not set to one, as this gives us some valuable information about the generalization ability of the network.

4.6 Results from using Direct Inverse Learning on the simulator

As mentioned earlier, we use the same structures as in section 3.6. These three facial structures labeled Structure 1 to Structure 3, must still not be confused with their respective target positions, labeled Target 1 to Target 4.

4.6.1 Structure 1 (A simplified eye brown)

Table 4.1 shows the result of using our neural network as a Direct Inverse Learning model on face Structure 1 without noise, compared with the error obtained by a random pull. As we see, the error is significantly lower for all four target expressions. The largest gain lies in Target 1 and 2, while the numbers from Target 3 and 4 are less impressive. At a first glance we might therefore conclude that the neural network is less precise on Target 3 and 4.

If we are to explain this, it is important to remember that the neural network do not use the error defined in chapter 2 as feedback. The error for the network is the target muscle strengths, and the input to the network is each point's position. The error spaces created for the target expressions in Structure 1 can be used in a much lesser degree to explain why the neural network behaves as it does. It is the internal error landscape of the neural network that is the key to the performance of the model, and this error space has got far too many dimensions to be visualized in three dimensions.

	Target 1	**Target 2**	**Target 3**	**Target 4**
Average of 100 iterations run 1	0.058	0.094	0.262	0.719
Average of 100 iterations run 2	0.053	0.089	0.234	0.722
Average of 100 iterations run 3	0.053	0.098	0.280	0.732
Internal model average	**0.055**	**0.094**	**0.259**	**0.724**
Random pull	0.58	0.67	0.73	0.98

Table 4.1: Error on suggested pull for face Structure 1 without noise. To see the deviation of the results, each row refer to the average value of 100 runs.

Prediction accuracy

To say something about the prediction accuracy of the network on behalf of the error obtained can be difficult, as it only tells us the relative distance of the structure to its target position. To get the structure's average distance in pixels from its target position, we have to multiply the error with the pixel distance from the initial position. The pixel distance from the initial position can be found by summing the distance of each point in a structure from the initial position to its target position. In addition, we have to use the error obtained without 1 as an upper limit. Table 4.2 shows the average distance in pixels from the predicted placement of the structure to its target position. As we can see by comparing the average error in percent from table 4.2 with the internal model average in table 4.1, the only target expression affected by not having 1 as maximum error is Target 4. Even though the largest error was obtained for Target 3 and 4, the accuracy of the network is lowest on Target 3 and 1.

	Target 1	Target 2	Target 3	Target 4
Distance from initial position	10252	4324	3333	199
Average error in percent without any upper limit on the error	5.5%	9.4%	25.9%	96.5%
Average sum of discrepancy in pixels	**564**	**406**	**863**	**192**

Table 4.2: Distance from the suggested position to each target in Structure 1, without noise

The distance graph

To understand some of the mechanisms responsible for these results, a three dimensional distance space graph might be helpful. By pulling each muscle a small amount at a time, we can create a graph that shows the structure's distance in pixels from the initial position, for all muscle strengths. The result is shown in figure 4.14. To understand the topology in the landscape, it is important to realize that each point's position is dependent on both the function affecting the muscles, and the functions affecting the skin. The limitation in accuracy caused by the resolution of the skin, will in addition to the the skin functions itself introduce low response for each point in the skin, especially when the muscle forces are low. Wrong muscle strengths suggested in the region of low muscle force, will thus not have large effect on the point's positions, thereby creating the flat region around Target 4. Target 3 on the other hand is in the steepest gradient of the entire error landscape. This is of course due to the target muscle strengths that are both 50%, which yields the most active region for both muscles.

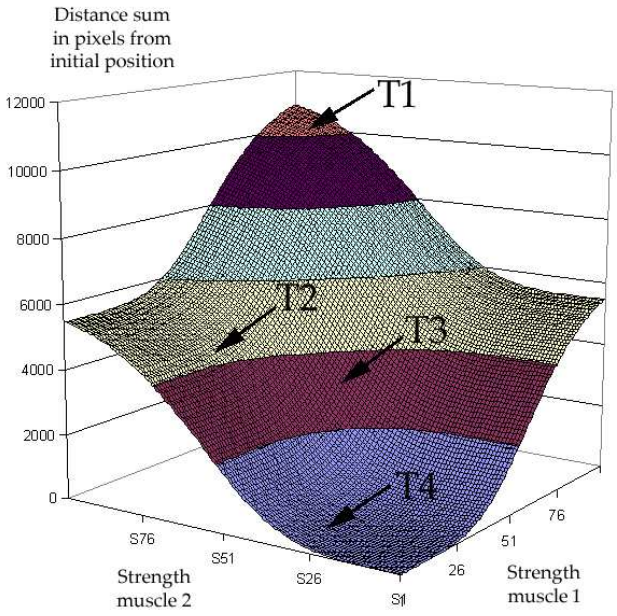


Figure 4.14: Structure 1's distance in pixels from its initial position (when both are relaxed). The arrows with the labels T1, T2, T3 and T4 shows muscle configuration and thus the distance for each of the four target expression to the initial position.

Accuracy vs. error

Deviation in muscle strengths from target positions lying in steep region will have a higher impact on the structure's distance. Visualised this way, we see that discrepancies in muscle strength are more severe for Target 3 than any other target. Predicted muscle strengths that are not entirely correct in this area will yield higher pixel distance than for any other target expression. Target 4 on the other hand, is the most forgiving one. Here small discrepancies in muscle strengths will yield low total pixel distance.

Nonetheless, as the error is measured relative to the original distance, the error becomes highest for Target 4, even though the network is most accurate at this position.

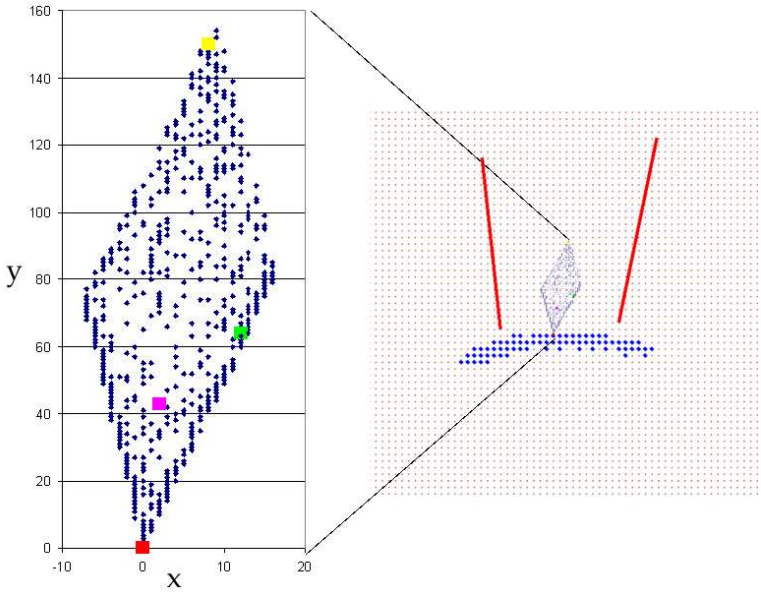


Figure 4.15: Positions of target point no. 26 in face structure 1 given 1000 random pulls. The initial position is in the bottom of the graph, marked red. The x-axis denotes the movement of pixels in the x-axis direction, and the y-axis in the y-axis direction. Point no. 26's position for the four target expression is marked in colour. Target 1 is yellow, Target 2 is green, Target 3 is pink and Target 4 is red. As we can see point no. 26 is not moved for Target 4, thus the red colour in the bottom of the graph.

Density of training examples

The non linear muscle strength has another effect on the network that might explain some of the differences in error obtained. Each training example for the network is made by random. Given an even distribution of muscles strengths, less training examples will be in the region where the muscles are most active, as this is the region of steepest gradient. This means that the neural network will have more training examples on the extremity of the muscle configurations than in the middle. Figure 4.15 visualises this. Figure 4.15 show the resulting x and y axis positions of target point no. 26 in face Structure 1. Point no. 26 is situated approximately in the middle of the eye brown between the muscles. The muscles are pulled randomly 1000 times, and the resulting position is plotted as a blue square. The initial position of the point is the bottom of the figure, while all other points are placed at their resulting x and y axis position relative to the initial position. As we can see, all targets except Target 3 are in areas with high density of training examples. It should therefore be easier for

the neural network on average to achieve more accurate muscle strengths if the target positions of the points are in these areas, further explaining why the discrepancy of Target 3 is so big. The same logic can be used to explain the difference between Target 1 and 2. Target 1 lies in a steeper region than Target 2, making it more sensitive to deviation from the target muscle strength. Their pixel error is between Target 3 and 4, which also seems reasonable when looking at their target muscles strengths and the number of training examples they receive.

Noise

Table 4.3 shows the result of noise in the simulator. As we can see, introducing noise has almost no effect on the results. This can at first seem a little strange as the neural network base its calculation on the exact position of each point. On the other hand, suppose the network produced the exact muscle strength needed to reach each target position. Knowing that Target 3 is the target which is most sensitive to noise, we could calculate the result of introducing noise. The maximum noise introduced could alter each muscle force by 2.5%. Looking at figure 4.14, we see that the maximum pixel discrepancy would become approximately 300. Compared to the pixel difference of 863 obtained by Target 3, we realize that the noise introduced by the muscles are less important than other mechanisms for the inaccuracy in the inverse model.

The feedback noise on the other hand can be quite large, but was only introduced occasionally. The probability for feedback noise is set to 5%, giving on average 2 training examples with feedback noise. This is obviously not enough to disrupt the generalization of the network, and hence has low impact on the models performance.

	Target 1	**Target 2**	**Target 3**	**Target 4**
Average of 100 iterations run 1	0.061	0.097	0.256	0.730
Average of 100 iterations run 2	0.051	0.088	0.233	0.708
Average of 100 iterations run 3	0.054	0.101	0.275	0.751
Internal model average	**0.055**	**0.095**	**0.255**	**0.730**
Random pull	0.58	0.67	0.73	0.98

Table 4.3: Error on suggested pull for face Structure 1 with noise

Neural networks and random noise

The error measure, and the way neural networks are trained, also makes them quite insensitive to random noise. This is easy to visualize using a simple graph. Suppose a neural network is to output a linear function based on some input training data. If no noise were present during training, all points would stay on the same line. However, if random noise is introduced, the best the network could do is to find an average over the training examples. This can be easily seen by looking at the error measure for a neural network:

$$E(\vec{w}) = \sum_{u} \frac{1}{2}(t-o)^2$$

Where u is the number of training examples. Hence, the neural network will try to minimize the sum of errors for all training examples, and the function suggested by the network will end up fairly close to the original line. Figure 4.16 shows this relationship.

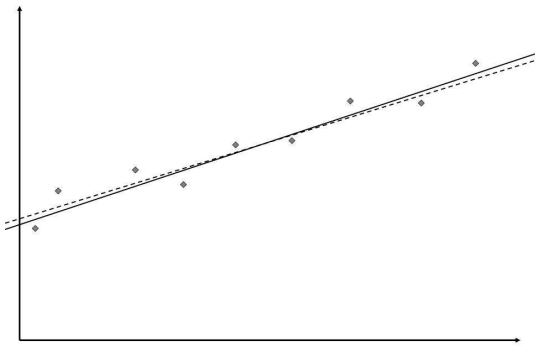


Figure 4.16: The effect of random noise on a neural network. The black line is the target function, while the stippled line is the function suggested by the network on behalf of the points (training examples affected by noise).

4.6.2 Structure 2 (A straight wrinkle)

Looking at the results from the second face structure shown in table 4.4, we get to very much the same the same pattern as for Structure 1. The inverse model performs a lot better than random pull for the first three target positions, but its performance is reduced for Target 4. However, it is worth noting that the network configuration for Structure 2 has 20 hidden nodes, and not 40 as for Structure 1. The reason is that the network in structure 1 has 142 inputs, while Structure 2 has only 46, easing the computational task.

	Target 1	**Target 2**	**Target 3**	**Target 4**
Average of 100 iterations run 1	0.051	0.102	0.356	0.901
Average of 100 iterations run 2	0.050	0.094	0.325	0.919
Average of 100 iterations run 3	0.053	0.098	0.314	0.895
Internal model average	**0.051**	**0.098**	**0.332**	**0.905**
Random pull	0.62	0.57	0.69	1.00

Table 4.4: Error on suggested pull for face Structure 2 without noise

Accuracy

Looking at the network's ability to generalize on behalf of the training examples, we see lot of similarities from Structure 1, shown in table 4.5. As for the first facial structure, the target position with highest error in pixels is the one with the most active muscles. However, Target 4 is more affected by the removal of the upper error limit than any of the target expressions in Structure 1. This might be explained by the very small distance in pixels from the initial position.

	Target 1	**Target 2**	**Target 3**	**Target 4**
Distance from initial position	3131	1302	869	12
Average error in percent without any upper limit on the error	5.1%	9.8%	33.0%	180.0%
Average sum of discrepancy in pixels	**160**	**128**	**287**	**22**

Table 4.5: Distance from the suggested position to each target in Structure 2

Noise

Table 4.6 shows the error with noise added. As we see, noise still have no effect on the overall performance, even though it might seem like the error increases when we remove the upper limit in Target 4. This seems reasonable when we know that the target positions is very close to the initial position, and small discrepancies in the suggested value yields high error. As we can see, the error obtained with and without noise in Target 4 is approximately 0.9, given 1 as the maximum error. This indicates that only a few of the trials yields a lower error. Therefore, the error measure with 1 as an upper limit is not affected by noise, even though noise affects the preciseness of the network on these small distances.

	Target 1	**Target 2**	**Target 3**	**Target 4**	
	Limit	Limit	Limit	Limit	No limit
Average of run 1	0.055	0.096	0.340	0.932	2.807
Average of run 2	0.060	0.107	0.326	0.893	1.736
Average of run 3	0.052	0.098	0.316	0.932	2.112
Internal model	**0.056**	**0.100**	**0.327**	**0.919**	**2.218**
Random pull	0.62	0.57	0.69	1.00	

Table 4.6: Error on initial pull for face Structure 2 with noise

4.6.3 Structure 3 (Two wrinkles with opposing muscles)

We saw in chapter 3 that the third face structure created the hardest challenge for the algorithms. For the simplest target expression, the error obtained by Stochastic Search increased from 0.01 in Structure 1 to 0.18 in Structure 3. For the inverse model, the error for Target 1 in Structure 1 and 3 is almost the same. As mentioned earlier, the mechanisms training the neural network are totally different form the mechanisms controlling the algorithms, and the same relationship is therefore no longer present. Looking at the results in table 4.7, we see very much to the same patterns as in Structure 1 and 2.

	Target 1	**Target 2**	**Target 3**	**Target 4**
Average of 100 iterations run 1	0.051	0.132	0.625	0.948
Average of 100 iterations run 3	0.058	0.143	0.625	0.933
Average of 100 iterations run 3	0.060	0.129	0.684	0.951
Internal model average	**0.056**	**0.135**	**0.645**	**0.944**
Random pull	0.62	0.55	1.00	1.00

Table 4.7: Error on suggested pull for face structure 3 without noise

Accuracy

Surely, the two last targets constitutes a greater challenge, but related to the generalization ability of the network, shown in table 4.8, the results are close to the same as for Structure 1 and 2. We see that the network reduces the error on the two easiest targets substantially, but it gives less improvement for Target 3, and almost no improvement for Target 4. This seem to further strengthen our theory the network seem to correct large deviations, but is less useful on small errors.

	Target 1	**Target 2**	**Target 3**	**Target 4**
Distance from initial position	2060	868	53	11
Average error in percent without any upper limit on the error	5.6%	13.5%	81.2%	398.1%
Average sum of discrepancy in pixels	**115**	**117**	**43**	**44**

Table 4.8: Distance in pixels for each target it structure 3 with noise

Noise

Table 4.9 shows the performance with noise added. Again we see that noise have little or no effect on the network, except from the values on Target 3 and 4, if there is set no upper bound on the error. This could again be explained by their very small distance to the initial position, where small deviations introduced by noise will have an effect on the error.

	Target 1	**Target 2**	**Target 3**		**Target 4**	
	Limit	Limit	Limit	No limit	Limit	No limit
Average of run 1	0.056	0.142	0.599	0.886	0.932	4.957
Average of run 3	0.062	0.128	0.658	1.224	0.949	5.659
Average of run 3	0.061	0.143	0.634	1.020	0.936	5.234
Internal model	**0.060**	**0.138**	**0.630**	**1.043**	**0.939**	**5.283**
Random pull	0.58	0.55	1.00		1.00	

Table 4.9: Error on initial pull for face structure 3 with noise

4.7 Performance summary

Accuracy

As we can see, introducing an inverse model reduces the error compared to random pull substantially if the target position is far away, but show less useful for small distances. The noise added in the simulator doesn't seem to have any noticeable impact on the overall performance. However, if the target expressions are situated close to the initial position and there is no upper limit on the error measure, we are starting to see the effect of noise. In contrast to the algorithms, the inverse model also seem to have close to the same performance on all facial structures. The computational challenge for the network is not the same however.

Computational challenge

Even though the first face structure has only got two muscles, the number of points in the structure is 71. The total number of inputs to the network is consequently 142. Face structure 1 therefore becomes the greatest challenge for the network, as can be seen from the number of hidden nodes. However, looking back at the two last facial structures, one realize that the muscles in structure 2 and 3 has almost no effect in the Y-axis direction. Half of the inputs to the networks in the two last structures are therefore inactive, or at least close to inactive. When we know that the two latter networks needs 20 hidden units, but only got approximately 25 varying inputs, we see that the computational task per input has increased from structure 1. They need almost one hidden unit per input to find a good approximation for their output. If the same structures were created with an angle relative to the Y-axis, we could suspect the computational task per input to increase even more.

Speed

Even though one of the premises in this thesis is that computational power in the future will increase, the amount of calculation needed for this kind of problem is considerable. On a 1.5GHz pentium M processor, the time needed for training the network one time is approximately 1,5 minutes. Increasing the number of hidden nodes yield even higher computational times.

Conclusion

There is no doubt that the inverse model implemented finds a correlation between each point's position and the muscle strength causing this state. However, the errors are still far from zero, indicating that an android based on Direct Inverse Learning should not base its ability to perform corrections on the inverse model alone.

In combination with a optimization algorithm, the internal model could be used to guide the algorithm by setting restrictions on the size of the error space, and suggest a initial start position. This could reduce the number of online trial substantially as the algorithm probably would start off by performing corrections in an area close to the global minimum of the error landscape.

Event though combining the inverse model with an optimization algorithm is a performance gain from only using algorithms, finding other means to increase the accuracy and improve the speed of the inverse model could be of great value.

4.8 Ways to improve the performance

4.8.1 Speed

Reducing the size of the network

In contrast to the algorithms, the neural network uses the positions of each point to create a model of the face. As we have shown, the complexity for the network increases with the number of input nodes. As mentioned in section 3.8, focusing on fewer points that are more important would be an easy way to go in order to reduce the complexity of the network, thereby easing the computational challenge.

Alignment of coordinate system

Even though the computational challenge might increase if structure 2 and 3 were created with an angle relative to the Y-axis, the problem should remain the same. This therefore tells us that using a coordinate system that is defined relative to the movement, or alternatively has fewer dimensions can be used to reduce the computational task. This might be a way to go if the movement of the structure in focus shows a simple movement pattern.

The learning algorithm

Back Propagation is based on gradient descent, and may often require ten thousands of iterations of weights updates to converge. There exists other methods to descend the error space that has shown to be more effective [31, 14]. In this thesis the speed requirement was low, but for a real world implementation, alternative learning algorithms should be considered.

4.8.2 Accuracy

There is also space for improvement in the tweaking the implemented network, such as using several hidden layers instead of one. While a network of one hidden layer can represent any continuous function, a network using several hidden layers can represent any function with arbitrary accuracy [32]. This might improve the performance in our simulator. However, as the network seems to converge to a relatively low error, we have reason believe that there are other mechanisms responsible for the discrepancies obtained in the simulator that are more important to solve.

4.8.3 Different internal model

Even though it is tempting to blame the neural network for the error obtained in the simulator, we believe the reason for this divergence lies to a larger degree in the nature of the inverse model implemented.

A common definition of a function is: *A function is a binary relation, f, with the property that for an element x there is no more than one element y such that x is related to y*[35]. Looking back at figure 4.14 we see that at small muscle strengths, many muscle configurations yield the same facial position. In effect, this means that the Direct Inverse Learning method is not trying to map a function. To see the consequences of this, it might be illustrative to use a simplified example.

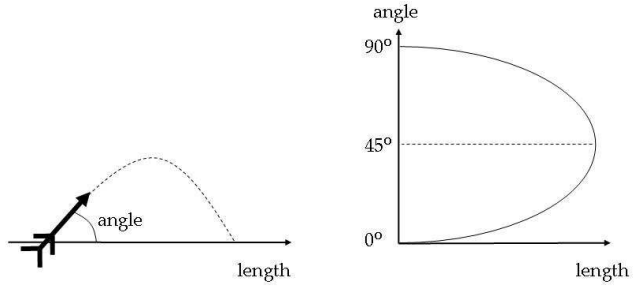
Simplified example

Figure 4.17(a) shows a one degree of freedom archery problem. An archer shoots an arrow with an angle relative to the ground between 0-90 degrees. Suppose the archer performed this exercise many times. We could plot the length traversed by the arrow as a function of the angle used. Figure 4.17(b) shows this relationship. As we see, for each length traversed by the arrow, there are two angles that yield that distance.

Remember that an internal model trained by Direct Inverse Learning uses the input output pairs in reverse. In this case, the input to the model is therefore the length traversed, and the output is the angle. This implies that an inverse model will see two different targets outputs for any given input.

Effect of error function

Going back to the neural network implemented in our simulator, the error of the network is given in equation 4.3. To see the effect of this error function, we pretend that the network only gets two training examples, namely 30 and 60 degrees. If the network corrected the weights in a matter that reduced the error for 30 degrees to zero, then the error when entering 60 degrees would be substantial. The Back Propagation algorithm would therefore descend this error space, and end up with 45 degrees, yielding the lowest error. If a least squares cost function is used, i.e. equation 4.3, then the internal model will output a value which is the average of the two targets.



(a) An archer shooting an arrow (b) The length traversed as a function of the angle

Figure 4.17: One degree of freedom archery problem

Many to one systems

In many real world applications, inverse models are one to many systems [20]. That is, there are many ways to go in order to reach a desired goal state. An android face is no exception. The skin and grimaces are functions of the force exerted by muscles and other factors. For each muscle force applied to the face structure, there is only one face expression, but this doesn't hold the opposite way [1]. That is, to create a specific face expression, there are several ways to pull the muscles. Going back to the simulator, this is the same problem encountered in the flat region of figure 4.14. Many muscle configuration yield the same skin position. The problem gets even worse if the muscles are pulling in opposite directions, as there are almost an infinite number of solutions to the problem. In a human face there are few muscles acting in exactly the opposite direction, but even muscles in the face with different angles can make up a problem. This can easily be seen by decomposing the force exerted by each muscle. Depending on the muscles relative angle to each other, there will always be a factor that yields opposing forces.

Reducing the number of "one to many system" instances

There exists ways to reduce the number of "one to many system" instances. One thing that could easily be implemented is the use additional factors such as the sum of forces exerted on the simulator. The task could therefore be to find the muscle strengths yielding a particular structure position and sum of muscle strengths.

[1] Unless, all atoms in the skin are used as feedback

Other means such as using more points in the skin could also used. However, even though it might be tempting to try and find a good inverse model by using Direct Inverse Learning, finding other schemes for creating internal models could turn out be more rewarding.

Chapter 5

Other ways to create internal models

Better internal models

As we have seen, internal models could be used to reduce the number of online trials. However, the precision of our internal model using Direct Inverse Learning turned out to be far from perfect. Seeing the limitations of Direct Inverse Learning we will in this chapter introduce two different strategies based on forward models for creating more accurate internal models.

Temporary noise

Even though we are able to find ways of correcting the imperfections in our internal model, the ability for these models to cope with temporary noise will depend on the timespan of the noise, and the time needed to update the model. We will therefore introduce the MOSAIC architecture [5, 7, 1] that could be used to cope with temporary noise and create large and advanced internal models.

Creating a continuously adaptive android face

Based on these methods, we will at the outset of the chapter introduce a new scheme of how to create an adaptive android face. This new model can in theory create an android that is continuously adaptive under all scenarios, thereby enabling natural mimic.

5.1 Forward models

Going back to the problem of creating an internal model, it is important to look closer at the computational challenge given to an inverse model. In essence, an inverse model is a forward model with predictive capabilities. This can easily be seen by looking at problem encountered in our simulator.

- For the inverse model in our simulator to come up with a correct prediction of how to pull the muscles, it had to mimic the physics of the muscles and the skin. This correlation can be looked upon as the forward model of the face dynamics, so the inverse model had to incorporate a forward model.

- Secondly the inverse model had to "search" its own forward model in order to come up with a suggestion of how to pull the muscles.

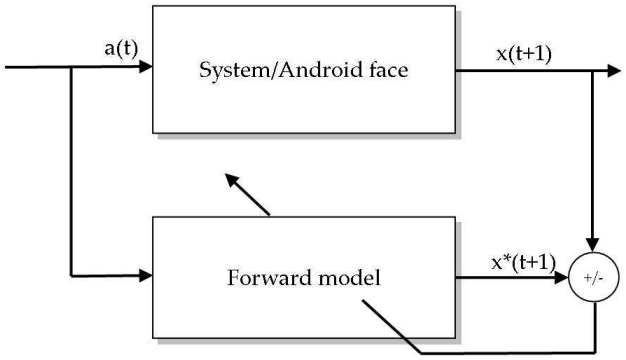


Figure 5.1: Forward model learning. The difference between x(t+1) and x*(t+1) is used to train the forward model

In contrast to the our inverse model, the relationship in the forward model is a function by definition. For any muscle strength, there exists one, and only one position for each point[1]. This difference between forward and inverse models is in fact present in many problems encountered by motor control [20].

The best way to create an internal model could therefore be through constructing exact forward models. Luckily these models are easier to train than inverse models, and we will explain how in the next section. However, creating an exact forward model of the skin dynamics is only a halfway solution to the original problem, namely how hard to pull each muscle in order to come up with the desired face expression. We introduce two methods that can be used in combination with a forward model to come up with a desired motor command, namely:

1. Searching a forward model
2. Distal Supervised Learning [18]

Forward model learning

The principle behind training a forward model is exactly the same as for Direct Inverse Learning, except from the fact that we don't need to inverse the input output pairs. This difference becomes critical as this is where the many to one problem in Direct Inverse Learning is introduced; *If y is a function of x, then x is not necessarily a function of y.*

Any system that can monitor the input and output of its own apparatus can also be used as a self induced supervised learner for the target function in forward model learning. A feed forward neural network trained by Back Propagation could be used as a forward model. In the case of our simulator, the network could take the muscle contraction as input, and adjust the weights so that the output would be concise with each point position. Figure 5.1 shows the principles behind forward model learning.

5.1.1 Searching a forward model

Given a successful implementation of a forward model, i.e. a model that is able to predict the skin position given a muscle contraction within an acceptable error limit, one could use this model to search for the correct muscle strength in order to derive the target face expression. Searching such an error space will give close to the same challenge as the search performed in chapter 3. The algorithms would have to suggests

[1] In the absence of noise.

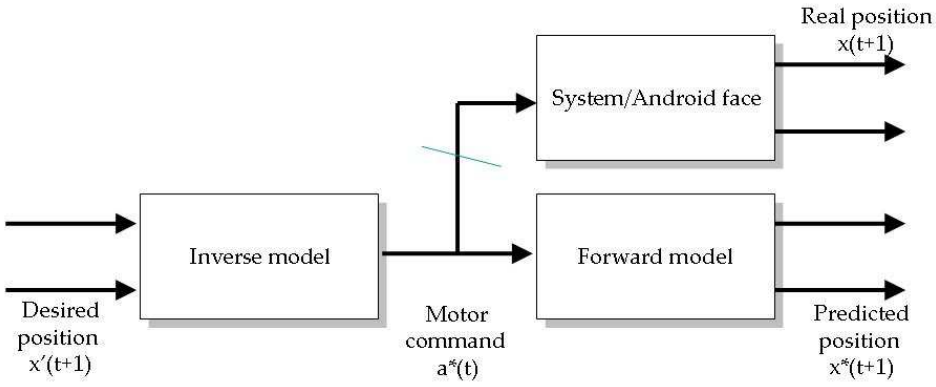


Figure 5.2: The architecture of distal supervised learning. The predicted muscular command a*(t) is only sent through the system if the predicted position is the same as the desired position, i.e. x*(t+1) = x'(t+1)

muscle strengths and give these as input to the forward model. The output of the forward model would provide the feedback error necessary to perform the search.

In contrast to the challenge for the algorithms in chapter 3, the algorithms searching a forward model do not need to have any restrictions on how many iterations they can perform. The search is done off line, and the time consuming nature of contracting the muscles are absent. Therefore, benchmarking such as CPU load, or time in milliseconds would be more important than the number of iterations.

Even though searching a forward model might be an easy and fast way to go for creating adaptive facial expression, Jordan and Rumelhart [18] has proposed a more advanced and maybe a more elegant solution to the problem.

5.1.2 Distal Supervised Learning

While forward models can be learned relatively straightforwardly by supervised learning, inverse models prove more problematic. Nonetheless, as we have seen, both types of models can be of great value to an organism. Distal Supervised Learning [18, 19, 20] uses both a forward and an inverse model to create an internal model. This configuration has two main advantages:

1. It addresses the problem of many to one systems encountered by Direct Inverse Learning
2. It enables continuous training of the models, increasing the reliability of the internal model.

The composition of Distal Supervised Learning is a little more complex than for Direct Inverse Learning, but the main difference becomes how error signals are created, and how they are used to correct the models. Distal Supervised Learning works by coupling an inverse and a forward model in series. Figure 5.2 shows the composition of Distal Supervised Learning.

Unity function

By looking at these two models as a composite system, we realize that if the system is perfectly trained, the input should be exactly the same as the output. The composite system should therefore yield a unity function. For the case of the simulator this would mean that the input (each point X and Y axis position) were turned into a muscular command by the inverse model, and these muscular commands would be propagated

through the forward model to produce the same X and Y axis positions as the input. If the predicted position is equal to the target position, the inverse model can send its command through the system, hopefully producing the desired target position.

Training

Training of DSL happens in two stages.

1. Training of the forward model

2. Training of the inverse model

As we have seen, the forward model can be learned directly by receiving the input to the system, and observing the output. Once this model is good enough (it doesn't even have to be perfectly trained), we can start the training of the inverse model. This is achieved by applying a specific input pattern on the composite system, and observing the output. If these values are not the same, the error can be used to correct the inverse model. The key to success is to hold the weights in the forward model constant, and this is why the forward model has to be trained first. By holding the weights in the forward model constant, the error can be back propagated all the way to the output of the inverse model, and from here, the error can start to correct the weights in the inverse model.

Error

It is worth noting that if the system after a successful training obtains an error when the system performs the action, then this error is a result of a flaw in the forward model. The real value obtained by the system can be used to correct the forward model, and thus reduce future errors.

We realize that even if the forward model is not perfect, the inverse model can be trained in a way that makes the entire composite system work perfectly. However, relying on imperfection in the forward model is not always a good solution, as this might lead to discrepancies when the final command is to be executed by the real system.

Conclusion

To summarize, DSL is in effect a combination of Direct Inverse Learning and searching a forward model. In contrast to DIL, Distal Supervised Learning finds only one correct solution to the problem while DIL takes the average over all correct solutions. DSL finds one correct solution either by having an inverse model that is trained to only find that one solution, or by searching the forward model by training the inverse model.

5.2 MOSAIC

So far in this thesis, we have omitted temporary noise, one of the four factors creating a need for adaptive facial behaviour. The MOSAIC model [5, 7, 1, 45] was introduced by D.M. Wolpert and M.Kawato in 1998. The model has two main advantages:

1. It addresses the problem of temporary noise

2. It can be used to create complex internal models

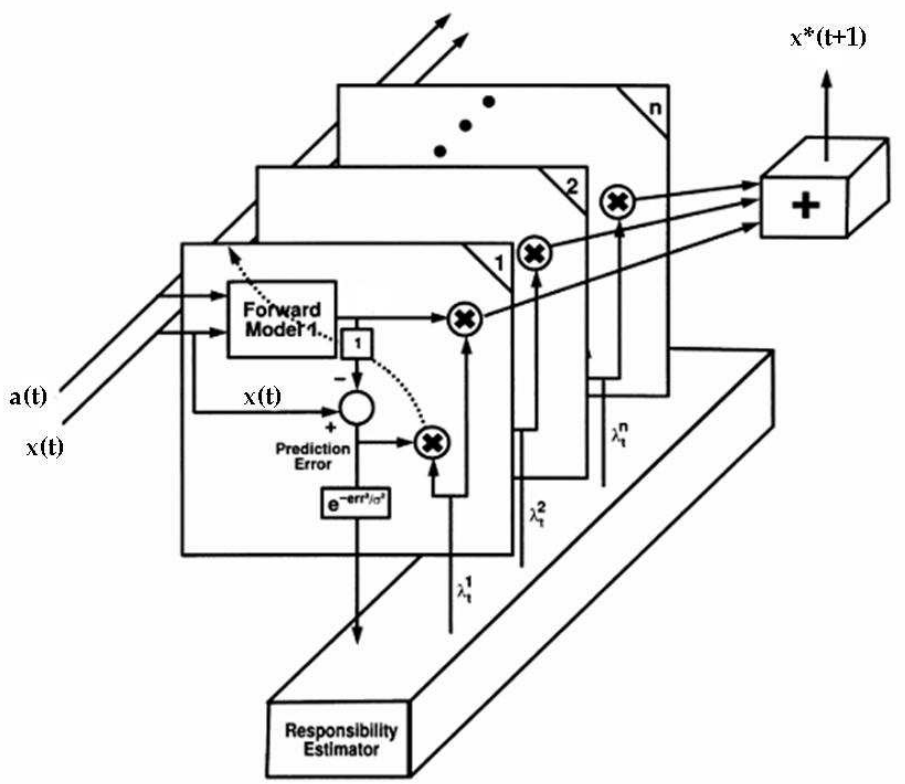


Figure 5.3: The MOSAIC architecture. x(t) is the current state of the system, e.g. the current position of the skin. a(t) is the muscular command proposed by a weighted sum of the inverse models. x*(t+1) is the weighted sum of the predicted next state as a result of the muscular command a(t) and the current state x(t). After the system has performed the action, the resulting new position can be compared with the next state predicted by each forward model. The forward model with the highest accuracy will receive the highest responsibility λ_t.

The MOSAIC architecture

The MOSAIC model allows individual internal models to participate in control without affecting the behaviour already learned by other modules. In the MOSAIC architecture, multiple paired inverse forward models predict in parallel the outcome of a motor command. The model with highest accuracy will be given greater amount of responsibility for future motor commands. Figure 5.3 shows a simplified diagram of the architecture behind the MOSAIC model.

The main contribution with the MOSAIC model is the way each forward inverse model pair is selected, used and trained. Instead of having only one model to predict and issue motor command at all times, the motor command can be looked upon as a merger of commands, based on each paired inverse-forward model's responsibility. Central to this process is the responsibility estimator.

Responsibility

Given a set of inverse-forward models pairs that predicts the outcome of an action, the model pair that has the highest predictability achieves the highest responsibility. This responsibility determines the amount of control given to the inverse-forward model pair. The final controls signal issued to the system can therefore be a weighted sum of several inverse models, based on their forward model predictive performance. The

same responsibility signal is also used to make adjustments to the internal models. If the system produces an error, then models with high responsibility will have to adapt to the error obtained, but models with low responsibility will be left more or less unaffected.

The MOSAIC architecture therefore allows individual models to participate in the learning and control without affecting the behaviour already learned by other modules. Such modularity can therefore reduce interference between what is already learned and what is to be learned, thereby both speeding up learning while retaining previously learned behaviours.

Temporary noise

In the context of an android face, the MOSAIC architecture could be valuable for an android to adapt to temporary noise in an efficient manner. By having several models for different contexts such as varying temperature, muscle fatigue or mechanical failure, the android could switch over to the most correct model at all times. The models in use would continuously be adjusted, while models not in use would be left unaffected, thereby enabling fast switching and reduced learning times when the environmental contexts change.

Complex internal models

In addition, the MOSAIC structure can be used to create a multitude of behaviours based on a limited amount of specialized models. By combining the output of different models, a large variety of contexts can be addressed, without having to take into account all possible variables. This will be clearer when looking at our proposed scheme for creating an android with an adaptive face introduced in the next section.

5.3 Divide and conquer, using a MOSAIC architecture

To justify the reasons for creating and analyzing the simple structures created in our simulator, we are going to propose a new scheme on how to create an android with adaptive facial behavior, consisting of:

1. Simple structures
2. The MOSAIC architecture based on forward models

We believe that by using these components, it is possible to create an android with adaptive facial behavior under all scenarios.

Simple structures

As we have seen, the computational challenge grows with increasingly complex structures. A natural way to cope with this challenge would therefore be to separate the face into regions under more or less local control. By dividing the face based on distinct muscular structures, the computational challenge for each region can be reduced while at the same time enable a high level of accuracy for each structure.

Say for example that the android wants to put on a smile. First it will look up the muscular command and resulting facial expression for a smile stored in its memory. Secondly, it will issue different muscular commands and target positions to each facial structure. All structures will pull its muscles accordingly, and read out the error. In the case of discrepancy between the target position and the issued muscular command, each structure becomes responsible for correcting its own structure properly, holding the complexity of each internal model to a minimum.

However, the problem with using this technique is that each structure will be affected by surrounding regions. Therefore, the physics in each structure will vary depending on the context. In other words, two different face expressions could demand two different internal models for each structure, as the external forces acting upon each region might be different for both facial expressions.

Adding the MOSAIC architecture

By using the MOSAIC architecture in each structure, one could address the problem introduced by using divide and conquer. This scheme could therefore be used not only to cope with temporary noise, but also to create a simple and robust architecture that will enable adaptive facial behavior under a variety of settings.

In the MOSAIC architecture, a limited amount of models in each structure could be combined to produce accurate internal models under a large variety of different contexts, i.e. different face expressions or other external factors. The output of the different MOSAIC models could hopefully be combined to create an accurate prediction of each structure, enabling correction of errors under a large variety of face expressions and other factors.

Self categorizing

In essence, what the MOSAIC model does is to categorize the external forces acting on one structure in several regions. External forces that are close to the same will gather in regions of high density. For each region of high density, one or a few models will be responsible for performing corrections.

If the MOSAIC architecture is to perform a prediction, it will combine several models if the external forces lay outside a region of high density. If the external forces lie in a region of high density, the model responsible for this region will perform the prediction.

When performing a prediction, the internal models will be affected by the error signal if they lie close to the region of interest, but can neglect the error signal if they are far away. Therefore the models can be trained to perform exact predictions in its own region, but they can also be combined to give predictions about intermediate solutions.

Conclusion

Even though using the MOSAIC architecture in combination with simple structures seem like a promising strategy, there are several questions that can be raised using our new proposed scheme.

How many internal models are needed for the MOSAIC architecture to provide accurate muscular commands given a large variety of face expressions? Will the muscular commands issued be accurate enough? What will be most efficient, using optimization algorithms or inverse models in the MOSAIC model?

We will not make an attempt to provide any solutions to these problems at this point without leaving this new scheme as a proposal for further work.

Chapter 6

Conclusion and proposal for further work

6.1 Conclusion

The main issue in creating android faces which resemble human appearance still lies in the materials. Up until now, no one has therefore focused on adaptive facial behaviour. However, we see a growing focus on creating android heads, and have therefore put adaptive facial behaviour on the agenda for the first time.

The original idea was to create an android head in hardware with Flexinol® wires as actuators. This android head was to be used as a tool to see some of the opportunities and limitations in this field. Unfortunately this implementation raised a number of problems. We therefore created a simple skin simulator instead. Our simulator has enabled us to show that:

1. Optimization algorithms can be used to create androids with adaptive facial behaviour, but not in social settings
2. If continuous feedback can be given from sensors in the skin, internal models could be used to increase the performance of optimization algorithms

Optimization algorithms

Even though our simulator is simplified to a great extent, some of the results might be useful in a real world implementation. We have shown that creating algorithms based on knowledge about the error space could be a smart way to go instead of relying on more general methods. Our new Push or Pull algorithm has turned out to be quite efficient, even though it has its weaknesses.

We have also realized the potential in using continuous feedback, such as a video stream. Our Push or Pull algorithm might with some modifications be implemented with success for continuous visual feedback. The result of using video feedback or algorithms made especially for this purpose is not tested in this paper. Nonetheless, we believe that this scheme can provide simple and efficient results, and we hope that this will be further investigated in the future.

Internal models

Given feedback from sensors in the skin, we have shown that internal models can be used to reduce the number of online trials. By analysing the results, we have found some flaws in our implementation of the internal model, thereby yielding imperfect results. Seeing its limitations and the most recent research on the subject, we have come up with a new scheme for creating androids with adaptive facial behaviour.

A new scheme for creating adaptive facial behaviour

Our new scheme is based on dividing the android face into smaller regions using a MOSAIC architecture for local control. If accurate feedback can be given from the skin, our model could in theory cope with all the reasons for having adaptive facial behavior mentioned in chapter 1. However, the model is highly theoretical, and a real world implementation could turn out to introduce several problems. To evaluate the usefulness of this scheme, we have come with several proposals for further work.

6.2 Proposals for further work

Search

Inspired by the simple implementation of using algorithms to create adaptive facial behaviour, we would like to see someone pursuing the task of using algorithms in combination with continuous visual feedback. We believe that video feedback would yield more efficient results than the quantified feedback used in this thesis. The focus should be on creating algorithms that corrected a face expression in the least number of seconds. To see the efficiency of new algorithms, Stochastic Search, and our simple Push or Pull algorithm could be used for benchmarking and comparison.

Scalability of forward models

As we have shown, creating exact forward models might be the best way to go in order to resemble human mimic. The main question will be how accurate they can be, the scalability, and how they should be controlled. If it turns out that forward models has no problems with scalability, then the reason for using the MOSAIC architecture proposed in or scheme disappears. It could therefore be interesting to create a simulator to test the scalability of forward models, and see their resulting accuracy given increased complexity.

Searching forward models

Even thought the Distal Supervised Learning approach devised by Rumelhart & Jordan is very elegant, searching a forward model can turn out to be even more efficient. A simulator comparing the CPU load on different forward models trained by DSL or searched by algorithms could give valuable clues as to which method is most efficient.

Creating a MOSAIC architecture

By creating a larger simulator having several regions that might have an influence on each other, the effect of using the MOSAIC structure could be analysed. The main question becomes how well this architecture will be in providing accuracy given different facial expression or varying contexts. By letting the simulator switch between one large forward model and the MOSAIC architecture, one could see if our suggested scheme has any advantages. Will there be any reduced computational challenge by using several regions with multi paired models, and how will the MOSAIC model affect the accuracy?

Bibliography

[1] Andrew Barto, *MOSAIC Model for Sensorimotor Learning and Control.* Neural Computation 13 (2001). Pages 2201-2220.
http://www.hera.ucl.ac.uk/sml/publications/index.html (Feb 06)

[2] Ben Krose, Patrick van der Smagt. An introduction to Neural Networks. (1998)
www.cs.unibo.it/~babaoglu/courses/cas/tutorials/Neural_Nets.pdf

[3] Android world.
http://www.androidworld.com/prod04.htm (Feb. 06)

[4] Bruce Alberts, Dennis Bray, Alexander Johnson, Julian Lewis, Martin Raff, Keith Roberts, Peter Walter. *Essential Cell Biology.* Garland Publishing (1997). Pages 304-308.

[5] Daniel M. Wolpert, Kenji Doya, Mitsuo Kawato. *A unifying computational framwork for motor control and social interaction.* Philosophical Transactions of the Royal Society 358 (2003). Pages 593-603
http://www.iis.ee.ic.ac.uk/yiannis/publications.html (Feb. 06)

[6] Daniel M. Wolpert, R. Chris Miall and Mitsuo Kawato. *Internal models in the cerebellum.* Trends in Congnitive Sciences Vol. 2, No. 9 (1998).
http://www.hera.ucl.ac.uk/sml/publications/index.html (Feb 06)

[7] Daniel .M Wolpert, Mitsuo Kawato, *Multiple paired forward and inverse models for motor control.* Neural Networks 11 (1998).
http://www.hera.ucl.ac.uk/sml/publications/index.html (Feb 06)

[8] David E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning.* Addison Wesley (1989).

[9] Davidson, P. R., Jones, R. D., Andreae, J. H., & Sirisena, H. R. *Detecting adaptive inverse models in the central nervous system.* Proceedings of 23rd International Conference of IEEE Engineering in Medicine and Biology Society
http://www.vanderveer.org.nz/research/publications (Feb 06)

[10] Differential Evolution Homepage
http://www.icsi.berkley.edu/~storn/code.html (Feb 06)

[11] Flexinol® Dynamic Alloys.
http://www.dynalloy.com/Links.html (Feb 06)

[12] Glenn Kenneth Kluthe. *Artificial muscles, Actuators for biorobotic systems.* (1999).
http://brl.ee.washington.edu/publications/Th024.pdf (Feb. 06)

[13] Hanson robotics.
www.hansonrobotics.com (Feb. 06)

[14] John Herts, Anders Krogh, Richard G. Palmer. *Introduction to the theory of neural computation.* Addison-Wesley Publishing Company (1991).

[15] Kurt Manal, Thomas S. Buchanan. *A one-parameter neural activation to muscle activation model: estimating isometric joint moments from electromyograms.* Journal of Biomechanics 36 (2003).
www.me.udel.edu/buchanan/PDF_Files/Manal%20&%20Buchanan%20JB%202003.pdf (Feb. 06)

[16] Melanie Mitchell. *An introduction to Genetic Algorithms.* MIT press (1996).

[17] Miall R. C & Wolpert D. M. *Forward models for physiological motor control.* Neural Networks 9 (1996): Pages 1265-1279.
http://www.hera.ucl.ac.uk/sml/publications/index.html (Feb 06)

[18] Michal I. Jordan, David E. Rumelhart. *Forward models: Supervised learning with a distal teacher.* Cognitive Science 16 (1992). Pages 307-354.
www-clmc.usc.edu/~cs542/jordan-CS92.pdf (Feb 06)

[19] Michael I. Jordan. *Computational aspects of motor control and motor learning.* Handbook of Perception and Action: Motor Skills, New York: Academic Press, 1996. .
www.snv.jussieu.fr/guigon/data/dea/Jordan96.pdf (Feb 06)

[20] Michael I. Jordan and Daniel M. Wolpert. *Computational motor control.* The Cognitive Neurosciences (1999).
www.hera.ucl.ac.uk/sml/publications/papers/JorWol99.pdf (Feb 06)

[21] Nanocyl.
http://www.nanocyl.com (Feb. 06)

[22] Ninitol Devices and Components.
http://www.nitinol.info (Feb. 06)

[23] Ph.D. Albert Mehrabian Homepage
http://www.kaaj.com/psych (Feb 06)

[24] R. C. Miall. *Connecting mirror neurons and forward models.* NeuroReport Vol. 14 No. 16 (2003)
prism.bham.ac.uk/pdf_files/Miall_2003_Neuroreport.pdf (Feb 06)

[25] Richard J. Mammone, Yehoshua Zeevi. *Neural networks theory and applications.* Academic Press Inc (1991).

[26] Stuart J. Russell, Peter Norvig. *Artificial Intelligence, A Modern Approach.* Prentice Hall (2003). Pages 111-115

[27] Stuart J. Russell, Peter Norvig. *Artificial Intelligence, A Modern Approach.* Prentice Hall (2003). Pages 116-119

[28] Stuart J. Russell, Peter Norvig. *Artificial Intelligence, A Modern Approach.* Prentice Hall (2003). Pages 736-749

[29] The Scottich Plastic & Rubber Association.
http://www.spra.org.uk/news22.html#six (Feb. 06)

[30] Tabu Search Web Site
http://www.tabusearch.net (Feb 06)

[31] Tom M. Mitchell. *Machine Learning.* McGraw-Hill (1997). Pages 81-127.

[32] Tom M. Mitchell. *Machine Learning.* McGraw-Hill (1997). Pages 249-273.

[33] University of Alberta, Shape Memory Alloys.
http://www.cs.ualberta.ca/~database/MEMS/sma_mems/sma.html (Feb. 06)

[34] Wikipdeia: "BRST algorithm"
http://en.wikipedia.org/wiki/BRST_algorithm (Feb 06)

[35] Wikipdeia: "Function (mathematics)"
http://en.wikipedia.org/wiki/Function_%28mathematics%29 (Feb 06)

[36] Wikipedia: "Genetic Algorithm"
http://en.wikipedia.org/wiki/Genetic_algorithm (Feb 06)

[37] Wikipedia: "Golden Section Search"
http://en.wikipedia.org/wiki/Golden_section_search (Feb 06)

[38] Wikipedia: "Gray code"
http://en.wikipedia.org/wiki/Grey_code (Feb 06)

[39] Wikipedia: "Hill Climbing"
http://en.wikipedia.org/wiki/Hill_climbing (Feb 06)

[40] Wikipedia: "Simulated annealing"
http://en.wikipedia.org/wiki/Simulated_annealing (Feb 06)

[41] Wikipedia: "Stochastic tunneling"
http://en.wikipedia.org/wiki/Stochastic_tunneling (Feb 06)

[42] Wikipedia: "Stochastic gradient descent"
http://en.wikipedia.org/wiki/Stochastic_gradient_descent (Feb 06)

[43] Wikipedia: "Optimization algorithms"
http://en.wikipedia.org/wiki/Optimization_algorithms (Feb 06)

[44] WorldWide Electroactive Polymer Actuators Webhub.
http://eap.jpl.nasa.gov (Feb. 06)

[45] Yiannis Demris, Bassam Khadhouri, *Hierarchical Attentive Mulitple Models for Execution and Recognition of Actions.* (2005).
http://www.iis.ee.ic.ac.uk/yiannis/publications.html (Feb. 06)